Java Regular Expressions: Taming the java.util.regex Engine

MEHRAN HABIBI

Apress™

Java Regular Expressions: Taming the java.util.regex Engine

ISBN (pbk): 1-59059-107-0

Printed and bound in the United States of America 12345678910

Technical Reviewer: Bill Saez

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steven Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Nate McFadden

Copy Editor: Nicole LeClerc

Production Manager: Kari Brooks

Production Editor: Laura Cheu

Proofreader: Linda Seifert

Compositor: Susan Glinert Stevens

Indexer: Kevin Broccoli

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

Practical Examples

'I took a speed reading course and read War and Peace in 20 minutes.
It's about Russia."
— Woody Allen

IN THIS CHAPTER, I'll think aloud as I solve several regex problems. This should provide some insight into the process of forming nontrivial regex solutions. Some of these examples are problems that I've found on a regex or Java newsgroup, and others were created for this chapter.

> **NOTE** The problems in this chapter take advantage of the `RegexProperties` class, which was defined in Chapter 4. `RegexProperties` is a class that extends `java.util.Properties` and allows you to load regex patterns from a properties file. The main advantage here is that the patterns don't have to be Java-delimited. Thus, you can actually use ***\d*** instead of ***\\d***. For more details on the `RegexProperties` class, please see Chapter 4.

Confirming the Format of a Phone Number

This first example confirms that a given phone number has the normal format for a U.S. phone number.

Before beginning in earnest, I have to make a decision: Do I want to write my own pattern or try to find an existing one? Normally, the first thing I would do is check some online regex resource, such as `http://www.regexlib.com`, to find an existing pattern. However, because this is a relatively simple pattern, and because I want to demonstrate the process of writing it, I'll create it myself.

Because I've decided to write the pattern myself, the first question I need to answer is, what is a phone number? I begin by working backward from some sample numbers. This is the pull technique described in Chapter 1. It requires that I take some actual data and try to pull the pattern out of it. Say my sample set is this:

614-345-6789

345-6789

345 6789

345.6789

3456789

(614)345-6789

6143456789

I pick the first phone number on the list as my pattern model. In examining it, I see a fairly obvious blueprint: three digits, a hyphen, three digits, a hyphen, and then four digits. That leads to the pattern ***\d{3}-\d{3}-\d{4}***. Table 5-1 shows the process of deriving the pattern.

Table 5-1. Pulling a General Regex Pattern from 614-345-6789

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 1	Nothing	Initial state	N/A	614-345-6789
Step 2	Replaced digits with ***\d***	To get a more generic description	***\d*** stands for any single digit	\d\d\d-\d\d\d-\d\d\d\d
Step 3	Replaced ***\d\d\d*** with ***\d{3}***	To produce a more succinct pattern	***\d{3}*** is exactly equal to ***\d\d\d***	\d{3}-\d{3}-\d\d\d\d
Step 4	Replaced ***\d\d\d\d*** with ***\d{4}***	To produce a more succinct pattern	***\d{4}*** is exactly equal to ***\d\d\d\d***	\d{3}-\d{3}-\d{4}

Of course, the pattern also has to accommodate numbers consisting of only seven digits, such as *345-6789*. At present, it can't, because it's modeled after data that has nine digits. Reconciling the pattern to do so leads to ***(?:\d{3}-)?\d{3}-\d{4}***. Table 5-2 shows the process of deriving the pattern.

*Table 5-2. Pushing **\d{3}-\d{3}-\d{4}** to Accommodate Seven-Digit Numbers*

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 5	Nothing	Initial state	N/A	\d{3}-\d{3}-\d{4}
Step 6	Grouped the leftmost ***\d{3}-*** as a single entity that might or might not exist.	To treat ***\d{3}-*** as a single entity that might or might not exist.	Any part of a pattern can be subgrouped.	(\d{3}-)\d{3}-\d{4}
Step 7	Made ***(\d{3}-)*** optional by producing ***(\d{3}-)?***	So that users can omit area codes.	Adding ***?*** after a group makes it optional.	(\d{3}-)?\d{3}-\d{4}
Step 8	Added a ***?:*** inside the opening ***(*** of ***(\d{3}-)*** to produce ***(?:\d{3}-)?***	It makes the expression more efficient. Non-capturing groups require less memory.	Adding ***?:*** inside a group makes the group noncapturing.	(?:\d{3}-)?\d{3}-\d{4}

Now the pattern will accept any seven or ten digit sequence of numbers, as long as they are grouped into sets of threes and fours, and separated by hyphens.

A Brief Digression on Working with Regex in Java

If I were programming in Perl, the next natural step would probably be to account for punctuation in the candidate, deal with an opening parenthesis that might or might not be there, and so on. Of course, this would most likely be addressed in the pattern itself.

But I'm not using Perl; I'm using a full featured, object-oriented language that's been designed to deal with nuisances while remaining clear. I decide to take advantage of that by scrubbing the data, and relying more on programmatic logic and less on regex wizardry.

Next, I use `String.replaceAll` to remove all punctuation and spacing from the phone number, as follows:

```
String scrubbedPhone = phone.replaceAll("\\p{Punct}|\\s","");
```

This replaces any and all punctuation or space characters with a zero-length string.

> **NOTE** ***\p{Punct}*** is a POSIX, U.S. ASCII predefined class that matches any punctuation character. Specifically, it matches ***!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~***. It was introduced in Chapter 1, in Table 1-12.

This way, I can count on the phone number being in the form *6143456789* or *3456789*. This is great news, because I can now simplify the pattern even further, as shown in Table 5-3. By breaking the process down into a separate step, I've decreased the amount of complexity that will go into a given pattern.

*Table 5-3. Removing References to - from **\d{3}-\d{3}-\d{4}** to Accommodate the Data Scrub*

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 9	Nothing	Initial state	N/A	(?:\d{3}-)?\d{3}-\d{4}
Step 10	Removed all references to the character -	This scrubbing guarantees that I'll never have to deal with punctuation in the phone number. Thus, it would be a mistake to expect it.	Removing the - simply means that the pattern won't check for, or require, the existence of a hyphen.	(?:\d{3})?\d{3}\d{4}
Step 11	Treated ***(\d{3})*** as a single entity and checked for its existence one or two times; thus, replaced ***(?:\d{3})?\d{3}*** with ***(?:\d{3}){1,2}***	To make the expressions less verbose.	***(?:\d{3})?\d{3}*** means "three digits or six digits." ***(?:\d{3}){1,2}*** means exactly the same thing. Thus, they're logically equivalent statements.	(?:\d{3}){1,2}\d{4}
Step 12	Went back to using the pattern from step 10.	Although the pattern from step 11 is briefer, it's more difficult to read.	See previous.	(?:\d{3})?\d{3}\d{4}

Notice that I back off from a change made in step 11 in step 12. Although it's true that step 11 made the expression less verbose, it also made it more difficult to read. In this case, I'm willing to have the pattern be slightly longer, if it will also be easier to read and maintain. Thus, the heart of my code consists of two lines. The first is line 32, which strips out any and all punctuation from the candidate:

```
String tmp = phone.replaceAll("\\p{Punct}|\\s","");
```

The second is line 36, which applies the pattern:

```
boolean retval = tmp.matches(PHONE_PATTERN) // (\d{3})?\d{3}\d{4}
```

Listing 5-1 shows the full implementation.

Listing 5-1. Searching for a Phone Number

```
01  import java.util.regex.*;
02  import java.util.logging.Logger;

03  public class MatchPhoneNumber{
04     private static Logger log = Logger.getAnonymousLogger();
05     private static final String   PHONE_NUMBER_KEY="phoneNumber";
06     /**
07     * Confirms that the format for the given phone number is valid.
08     * @param phone a String representing the phone number.
09     * @returns true if the phone number format is acceptable.
10    */
11    public static boolean isPhoneValid(String phone){
12       boolean retval=false;
13           String msg = "\r\nCANDIDATE:" + phone;


14       //make sure the candidate has a shot passing
15       if (phone != null && phone.length() > 6)
16       {
17           //load the regex properties file
18           RegexProperties rb = new RegexProperties();
19           try
20           {
21             rb.load("../regex.properties");
22           }
```

```
23          catch(Exception e)
24          {
25                e.printStackTrace();
26          }

27          //scrub the phone number, removing spaces
28          //and punctuation. We could store this
29          //pattern in the regex.property file as well,
30          //but it's not really so complex that
31          //it's confusing when Java-delimited
32          String tmp = phone.replaceAll("\\p{Punct}|\\s","");

33          //extract appropriate regex pattern and run check
34          //in this case (\d{3})?\d{3}\d{4}
36          String phoneNumberPattern=rb.getProperty(PHONE_NUMBER_KEY);

37          //do the actual comparison
38          retval= tmp.matches(phoneNumberPattern);

39          //log for debug purposes
40          msg += ":\r\nREGEX:" + phoneNumberPattern;
41       }
42       msg += "\r\nRESULT:" + retval +"\r\n";
43       log.info(msg);
44       return retval;
45    }
46   public static void main(String args[]) throws Exception{
47     if (args != null && args.length == 1)
48        System.out.println(isPhoneValid(args[0]));
49     else
50        System.out.println("usage: java MatchPhoneNumber <phoneNumber>");
51    }
52 }
```

Even programmers who don't know anything about regex can follow the code, which speaks to the elegance of J2SE regex.

Is the extra verbosity justified? Would it have been better to simply write the regex in a single line for this particular case? This is the sort of decision you'll need to make on a case-by-case basis for your particular needs. In my opinion, it's better to err on the side of verbosity than to risk terse code.

Confirming Zip Codes

The next challenge is to provide a method that confirms zip codes for the United States. The method needs to accommodate punctuation, a space, or no delimiter at all between the five-digit and four-digit parts of the zip code. It needs to accommodate zip codes that are only five digits long. Suddenly, there's requirements creep: It now needs to validate zip codes for Canada, the United Kingdom, Argentina, Sweden, Japan, and the Netherlands as well.

The first thing I do is search the Web for patterns, starting at `http://www.regexlib.com`. This returns regular expressions for all of the countries previously mentioned. Next, I take those regular expressions and create entries in the `regex.properties` file, so I can use the `RegexProperties` class from Chapter 4.

The point of doing so, of course, is to externalize the expressions themselves and to avoid having to double-delimit special characters. I decide to use intelligent keys for the property keys. That is, I'm anticipating that I'll have access to the country code for each of these regex patterns. Therefore, I can define the property file keys based on that country code. For example, since the country code for Japan is *JP*, I define the key to the zip code pattern for Japan as *zipJP*. Listing 5-2 summarizes the entries made to the `regex.properties` file.

Listing 5-2. New Entries in the regex.properties File

```
#Japanese postal codes
zipJP=^\d{3}-\d{4}$

#US postal codes
zipUS=^\d{5}\p{Punct}?\s?(?:\d{4})?$

#Dutch postal code
zipNL=^[0-9]{4}\s*[a-zA-Z]{2}$

#Argentinean postal code
zipAR=^\d{3}-\d{4}$

#Swedish postal code
zipSE=^(s-|S-){0,1}[0-9]{3}\s?[0-9]{2}$

#Canadian postal code
zipCA=^([A-Z]\d[A-Z]\s\d[A-Z]\d)$

#UK postal code
zipUK=^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1} {0,1}[0-9][A-Za-z]{2}$
```

Finally, I write the code. The algorithm is to look up the appropriate regex for a given country given the appropriate country code, apply the pattern, and return true or false as appropriate. Listing 5-3 shows the code that does this.

Listing 5-3. Matching Zip Codes for Various Countries

```
01 import java.io.*;
02 import java.util.logging.Logger;
03 import java.util.regex.*;

04 /**
05 *Validates zip codes from the given country.
06 *@author M Habibi
07 */
08 public class MatchZipCodes{
09     private static Logger log = Logger.getAnonymousLogger();
10    private static final String ZIP_PATTERN="zip";
11    private static RegexProperties regexProperties;
12    //load the regex properties file
13    //do this at the class level
14    static
15    {
16        try
17        {
18            regexProperties = new RegexProperties();
19            regexProperties.load("../regex.properties");
20        }
21        catch(Exception e)
22        {
23            e.printStackTrace();
24        }
25    }

26    public static void main(String args[]){
27        String msg = "usage: java MatchZipCodes countryCode Zip";

28        if (args != null && args.length == 2)
29            msg = ""+isZipValid(args[0],args[1]);

30        //output either the usage message, or the results
31        //of running the isZipValid method
32        System.out.println(msg);
33    }
```

```
34     /**
35     * Confirms that the format for the given zip code is valid.
36     * @param the <code>String</code> countryCode
37     * @param the <code>String</code> zip
38     * @return <code>boolean</code>
39     *
40     * @author M Habibi
41     */
42     public static boolean isZipValid(String countryCode, String zip)
43     {
44         boolean retval=false;
45         //use the country code to form a unique into the regex
46         //properties file
47         String zipPatternKey = ZIP_PATTERN + countryCode.toUpperCase();

48         //extract the regex pattern for the given country code
49         String zipPattern = regexProperties.getProperty(zipPatternKey);

50         //if there was some sort of problem, don't bother trying
51         //to execute the regex
52         if (zipPattern != null)
53             retval = zip.trim().matches(zipPattern);
54         else
55         {
56             String msg = "regex for country code "+countryCode;
57             msg+= " not found in property file ";
58             log.warning(msg);
59         }
60         //create log report
61         String msg = "regex="+zipPattern +
62         "\nzip="+zip+"\nCountryCode="+
63         countryCode+"\nmatch result="+retval;
64         log.finest(msg);

65         return retval;
66     }
67 }
```

Outside of the comments and such, the real work in this method is done in three lines. Line 47 forms the proper key based on the country code:

```
47      String zipPatternKey = ZIP_PATTERN + countryCode.toUpperCase();
```

For example, `zipPatternKey` equals *zipUS* for the *US* country code. Next, line 49 extracts the relevant pattern based on that key:

```
49   String zipPattern = regexProperties.getProperty(zipPatternKey);
```

Line 53 actually compares the pattern against the key:

```
53              retval = zip.trim().matches(zipPattern);
```

The only regex change I made in this example was to make the actual pattern just a little more memory efficient and a little more lenient, as shown in Table 5-4. Specifically, leniency means that the pattern will accept any punctuation, a space, or no delimiter at all between the first five digits and the last four digits of a U.S. zip code. The pattern will also accept five digits as a sufficient U.S. zip code.

Table 5-4. Making the Zip Code Pattern More Lenient and Efficient

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 1	Nothing	Initial state	N/A	\d{5}(-\d{4})?
Step 2	Added ***?:*** inside the capturing group ***(-\d{4})*** to produce ***(?:-\d{4})***	To produce a more efficient pattern.	We don't need a capture here.	\d{5}(?:-\d{4})?
Step 3	Replaced - with ***\p{Punct}?*** to produce ***(?:\p{Punct}?\d{4})?***	Any punctuation—or no punctuation at all—can be used as a delimiter.	***\p{Punct}?*** is a superset of -, and it's optional, so the regex engine is now willing to accept any punctuation or no punctuation at all as a delimiter.	\d{5}(?:\p{Punct}?\d{4})?

Table 5-4. Making the Zip Code Pattern More Lenient and Efficient (Continued)

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 4	Added a ***\s?*** pattern to the list of acceptable delimiters between the five digits and the four digits of a U.S. zip code	Zip codes that use a space or empty string to separate the five digits from the four digits will pass.	A space between the five digits of a U.S. zip code and the following four digits is optional. This is simply a more lenient interpretation.	\d{5}(?:\p{Punct}?\s?\d{4})?
Step 5	Moved the ***\p{Punct}?\s?*** out of the noncapturing group	This improves readability. Optional subpatterns inside a optional noncapturing group can be hard to follow.	Logically, the two are equivalent.	\d{5}\p{Punct}?\s?(?:\d{4})?
Step 6	Surrounded the pattern with a beginning-of-line ^ tag and an end-of-line *$* tag	This increases matching speed. The more precise the pattern, the better it will perform.	All zip codes will be coming into the method as extracted strings. Thus, they'll always have a beginning of line and an end of line.	^\d{5}\p{Punct}?\s?(?:\d{4})?$

Because the regex patterns are externalized, they can be tweaked later to become more accommodating for the various regions. Better yet, more country codes can be added without requiring code changes: Simply add the appropriate entries to the `regex.properties` file.

The point here is that even using generic regex patterns found online, I still have a very Java-like flavor to the code. It's modular, adaptable, scalable, and clear.

Confirming Date Formats

In this example, I need a method that will validate a date format. The requirements are very explicit. Some sort of punctuation between the various date tokens is required, and a space isn't considered punctuation. The method should accept either two digits or four digits for the year, and either one or two digits for the day and month. I also need to make sure the date isn't in the future. I can expect the first date token to be the month, the second date token to be the day of that month, and the last date token to be the year. Thus, valid entries might be as follows:

11/30/2002

4/25/03

03-29/2003

11/30/1902

2/25-03

06#9/2003

Again, the first thing to do is search the Web. I find a few patterns that might work here. The first follows, and it's described as being very robust, dealing with leap years, and so on: ***^(?:(?:(?:0?[13578]|1[02])(\/|-|\.)31)\1|(?:(?:0?[1,3-9]|1[0-2])(\/|-|\.)(?:29|30)\2))(?:(?:1[6-9]|[2-9]\d)?\d{2})$|^(?:0?2(\/|-|\.)29\3(?:(?:(?:1[6-9]|[2-9]\d)?(?:0[48]|[2468][048]|[13579][26])|(?:(?:16|[2468][048]|[3579][26])00))))$|^(?:(?:0?[1-9])|(?:1[0-2]))(\/|-|\.)(?:0?[1-9]|1\d|2[0-8])\4(?:(?:1[6-9]|[2-9]\d)?\d{2})$***.

I decide to pass on it for now.

The second pattern I find is ***^\d{1,2}\/\d{1,2}\/\d{4}$***, which looks promising, but limits itself to four-digit years. It could work, but I would have to tweak it. Next, I come across ***((\d{2})|(\d))\/((\d{2})|(\d))\/((\d{4})|(\d{2}))***. At first glance, it looks like I might have to pull the second pattern toward the third.

I don't like the third pattern as is, because it uses a lot capturing groups that it doesn't really need. Noncapturing groups would do as well, and they would be more efficient. This immediately makes me a little suspicious. It's also more verbose than I had hoped for. Of course, it's possible that the verbose nature of the expression makes it more efficient, but I doubt that an author who was worried about efficiency would have left all of those useless capturing groups in there.

Whichever pattern I choose, I'll probably want to replace any and all punctuation within the candidate string with a character that's easy to work with. I would just get rid of all punctuation, but then I wouldn't know if a date such as *1111971* was referring to January 11, 1971, or November 1, 1971. Thus, I'm going to need a line of code like this:

```
String scrubbedDate = date.replaceAll("\\p{Punct}","@");
```

Here, I'll probably use the @ symbol as a replacement delimiter. It doesn't have any sort of special regex meaning, so it's easier to work with. Next, I'll need to write a pattern to capture the month, day, and year, and make sure that it constitutes a valid date.

Wait a minute—I wonder if there's an easier way. What if I used the `String.split` method around the punctuation and extracted the date from the remaining digits? Then I could just use straight Java code to validate the actual date. To do that, I'll need something like this:

```
String[] datetokens = date.split("\\p{Punct}");
```

This looks fairly easy, so I go with it.

My algorithm becomes the following: Split the date along punctuation marks, use it to create a `Calendar` object, compare that to today, and return true if the `Calendar` object is less than or equal to today. I can write the preliminary method signature as follows:

```
public static boolean isDateValid(String date)
```

Figure 5-1 shows the algorithm.

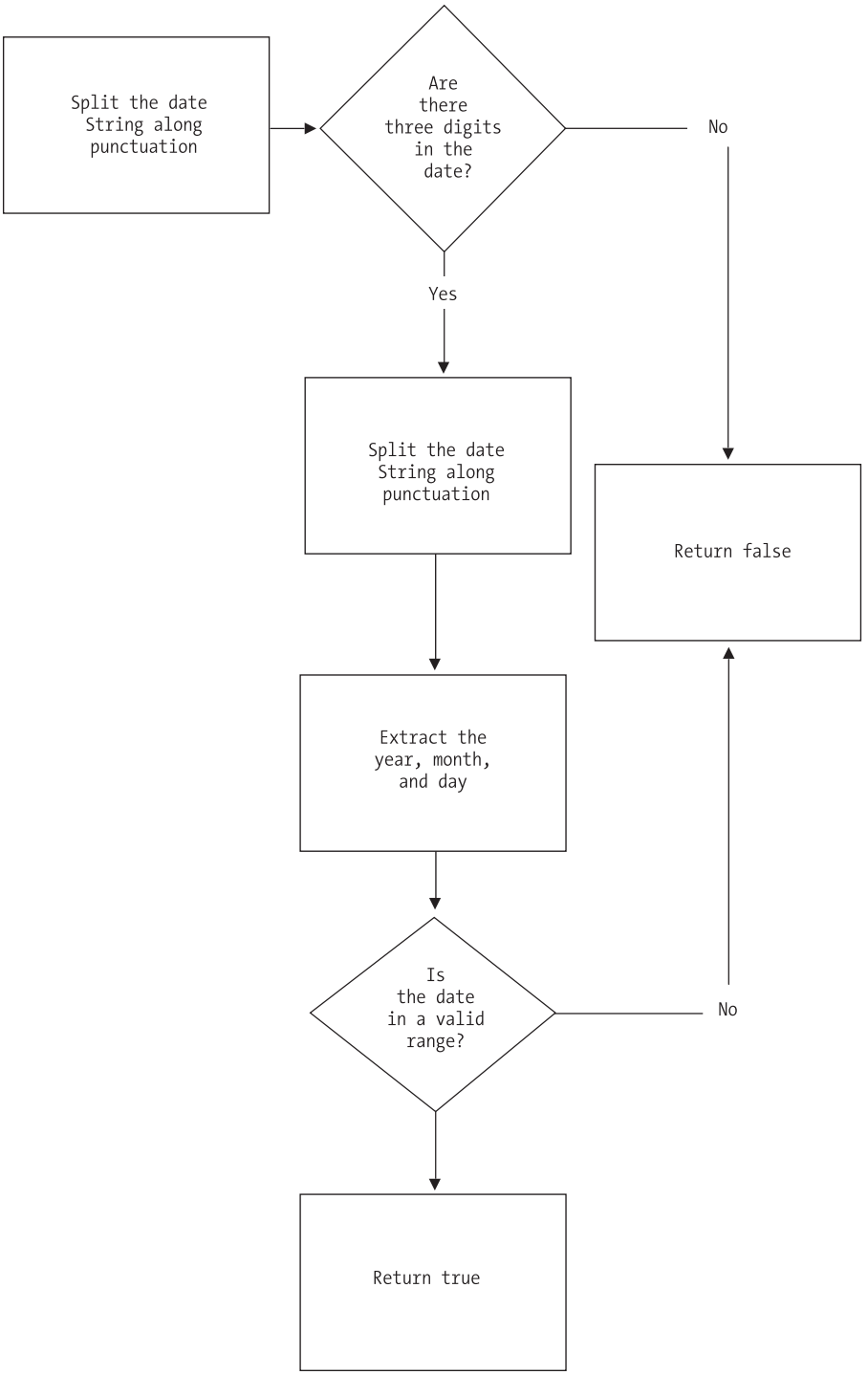

Figure 5-1. The algorithm for the isDateValid method

Listing 5-4 presents the full implementation.

Listing 5-4. Validating a Date

```
01 import java.util.regex.*;
02 import java.io.*;
03 import java.util.logging.Logger;
04 import java.util.GregorianCalendar;
05 import java.util.Calendar;

06 /**
07 *matches dates
08 */
09 public class MatchDates{
10 private static final String DATE_PATTERN = "date";
11 private static final String PROP_FILE = "../regex.properties";
12 private static Logger log = Logger.getAnonymousLogger();
13 public static int LOWER_YEAR_LIMIT = -120;

14    public static void main(String args[]) throws Exception{
15       if (args != null && args.length==1)
16       {
17         boolean b =isDateValid(args[0]);
18         log.info(""+b);
19       }
20       else
21       {
22         System.out.println("usage: java MatchDates dd/dd/dddd");
23       }
24    }

25    /**
26    * Confirms that given date format consists of one or two digits
27    * followed by a punctuation, followed by one or two digits
28    * followed by a punctuation, followed by two or four digits. Further,
29    * it actually validates that the date is less then today, and
30    * and not more then <CODE>LOWER_YEAR_LIMIT</CODE> =120 years in
31    * the past. This method even takes leap years and such into account
32    * @param the <code>String</code> date to be consider
33    * @return <code>boolean</code> true if
34    *
```

```
35   * @author M Habibi
36   */


37   public static boolean isDateValid(String date)
38   {
39     boolean retval=false;
40     date = date.trim();

41     //does the candidate have three digits? Otherwise
42     //the month, day, and year extraction below could
43     //throw a number format exception.
44     boolean hasThreeDigitSections =
45      date.matches("\\d+\\p{Punct}\\d+\\p{Punct}\\d+");

46     if (hasThreeDigitSections)
47     {
48        String[] dateTokens = date.split("\\p{Punct}");

49        if (dateTokens.length == 3)
50        {
51         //Java months are zero based, so subtract 1
52         int month = Integer.parseInt(dateTokens[0]) -1;

53         int day = Integer.parseInt(dateTokens[1]);
54         int year = Integer.parseInt(dateTokens[2]);

55         //in case a 2 digit year was entered
56         if (year < 100)
57           year += 2000;

58         //get boundary years
59         GregorianCalendar today = new GregorianCalendar();
60         //get a lowerLimit that is LOWER_YEAR_LIMIT less then
61         //today
62         GregorianCalendar lowerLimit = new GregorianCalendar();
63         lowerLimit.add(Calendar.YEAR, LOWER_YEAR_LIMIT);

64         //create a candidate representing the proposed date.
65         GregorianCalendar candidate =
66         new GregorianCalendar(year, month,day);
```

```
67          //check the validity of the date
68          if
69          (
70             candidate.before(today)
71           &&
72             candidate.after(lowerLimit)
73           &&//month could be off, say the user entered 55
74             month == candidate.get(Calendar.MONTH)
75           &&//day could be off, say the user entered 55
76             day == candidate.get(Calendar.DAY_OF_MONTH)
77          )
78          {
79              retval = true;
80          }
81       }
82      }
83      return retval;
84    }
85 }
```

The previous example deferred almost all of the heavy lifting to regular expressions. The code in Listing 5-4 uses regex for the split method. Otherwise, it's fairly conventional Java code. This doesn't mean the regex contribution is trivial—as a matter of fact, I would say it's critical. However, once the split method's regex contribution is assimilated, you're back in the comfortable world of Java.

Searching a String

This example searches a given string for the existence of a pattern and returns all of the matching strings. This is very easy code, but it's such a useful little program that it's worthwhile to demonstrate it.

First, I need to decide exactly what I mean by "return." Return what? In this case, I decide to return an `ArrayList` of matching `Strings`, because I want the `Strings` to be in the order in which they were found, and an `ArrayList` maintains the order in which elements were inserted. Also, I like the idea of returning a well-defined data structure, in case the client wants to, say, step through that structure and examine the data further.

I also decide that I want the client to be able to pass in `Pattern.compile` flags such as `Pattern.MULTILINE` and `Pattern.DOTALL`. It doesn't really cost me anything in the way of additional complexity, and it's a nice feature for the client. At this point, it's worthwhile to get a preliminary method signature written down. I come up with this:

```
public static ArrayList searchString(
   String content, String searchPattern, int flags
) throws IOException
```

Now I'm ready to start writing my method. My first pass looks like Listing 5-5.

Listing 5-5. First Pass at the searchString Method

```
01      public static ArrayList searchString(
02          String content,
03          String searchPattern,
04          int flags
05      )
06      throws IOException
07      {
08          ArrayList retval = new ArrayList();
09          Pattern pattern = null;

10         //compile the pattern
11         if (flags > -1 )
12         {
13             pattern = Pattern.compile(searchPattern, flags);
14         }
15         else
16         {
17             pattern = Pattern.compile(searchPattern);
18         }

19         //extract the matcher for the pattern
20         Matcher matcher = pattern.matcher(content);

21         //iterate through all of the matches, and add
22         //all relevant ones to the arrayList
23         while (matcher.find())
24         {
25             //extract the match and its position
26             String tmp = matcher.group();
27             //insert the matching string
28             //into the map.
29             retval.add(+ tmp);
30         }

32         return retval;
32     }
```

Listing 5-5 isn't terrible. It finds all the relevant matching substrings and returns them in order. I run a few sample tests and find that it works as expected. But it does leave something to be desired. It doesn't really tell me where the string was found, and it might be nice if it were overloaded, so the client isn't forced to pass in a flag if they don't need one.

I decide that for this generation, the client can live without the overloading. However, I do think the client has a right to ask for the position at which the matching strings were found. Thus, I modify the code so that it returns a `Map`. The `Map` will contain a key/value pair, which will use the byte position of each find (stored as a `String` or an `Integer`–I haven't decided which yet) and the matching substring as a value. Modifying the code, I come up with Listing 5-6. The only significant changes are on lines 7, 25, and 29. By the way, I decided to use a `LinkedHashMap` on line 8, because I wanted to preserve the order in which the matching `String`s were found. A `LinkedHashMap` is a J2SE 1.4 addition to the `Map` family that preserves the insertion order of elements.

Listing 5-6. Modified searchString Method Belonging in the RegexUtil Class

```
01  public static Map searchString(
02      String content,
03      String searchPattern,
04      int flags
05  )

06  {
07      Map retval = new LinkedHashMap();
08      Pattern pattern = null;

09     //compile the pattern
10     if (flags > -1 )
11     {
12         pattern = Pattern.compile(searchPattern, flags);
13     }
14     else
15     {
16         pattern = Pattern.compile(searchPattern);
17     }

18     //extract the matcher for the pattern
19     Matcher matcher = pattern.matcher(content);
```

```
20      //iterate through all of the matches, and add
21      //all relevant ones to the arrayList
22      while (matcher.find())
23      {
24          //extract the match and its position
25          int position = matcher.start();
26          String tmp = matcher.group();
27          //insert the matching string and position
28          //into the map.
29          retval.put(position+"",tmp);
30      }

31      return retval;
32  }
```

I decide to make the position a `String`, to make dealing with the output easier. I don't want to require the client to handle the keys too carefully, so `String`s will do for now.

Searching a File

Building on the previous example, I decide to provide a utility for searching the content of a file and returning all matching strings within that file. I'll use `FileChannel`s for the actual file I/O. Although a discussion of `FileChannel`s is beyond the scope of this book, in my opinion they're the best way to access files in Java.

My strategy is to use a `FileChannel` to open a file, read its content into a `String`, release the `FileChannel`, and then use the `searchString` method to parse the `String`. This is faster than reading through the file line by line and examining its content, though it is memory intensive. Listing 5-7 shows the code for doing this.

Listing 5-7. Reading in File Content

```
01      /**
02      * extracts the content of a file
03      * @param String fileName the name of the file to extract
04      * @throws IOException
05      *
06      * @return String representing the contents of the file
07      */
08      public static String getFileContent(String fileName)
09      throws IOException{
10          String retval = null;
```

```
11          //get access to the FileChannel
12          FileInputStream fis =
13            new FileInputStream(fileName);
14          FileChannel fc = fis.getChannel();

15          //get the file content
16          retval = getFileContent(fc);

17          //close up shop
18          fc.close();
19          fc = null;

20          return retval;
21      }

22      /**
23      * extracts the content of a file
24      * @param String fileName the name of the file to extract
25      * @throws IOException
26      *
27      * @return String representing the contents of the file
28      */
29      private static String getFileContent(FileChannel fc)
30      throws IOException{
31          String retval = null;
32         //read the contents of the FileChannel
33          ByteBuffer bb = ByteBuffer.allocate((int)fc.size());
34          fc.read(bb);

35          //save the contents as a string
36          bb.flip();
37          retval = new String(bb.array());
38          bb = null;

39          return retval;
40       }
```

Next, I need to provide a method that will load the file, search it, and return the results. Given the two previous methods, this becomes fairly easy, as shown in Listing 5-8.

Listing 5-8. Opening a File, Searching the File, and Returning the Results

```
01      public static Map searchFile(
02          String file,
03          String searchPattern,
04          int flags
05      ) throws IOException
06      {
07         String fileContent = getFileContent(file);
08         Map retval = searchFile(fileContent,searchPattern,flags);
09         return retval;
10    }}
```

I take the program out for a spin and compare it to grep. To be honest, it seems to lack a bit in the comparison. The grep program returns the entire line of a matching token, whereas this method only returns the matching token. That's not terrible, because the client could request the entire line by using the correct regex pattern. But it's not really as friendly as it could be, especially for the average user.

I decide to "pad" the pattern to capture an entire line, assuming that the original search pattern has no punctuation, and thus no regex, in it. Listing 5-9 shows my modified `searchFile` method.

Listing 5-9. Modifying the searchFile Method to Make It Friendlier

```
01  public static Map searchFile(
02      String file,
03      String searchPattern,
04      int flags
05  ) throws IOException
06  {
07     String fileContent = getFileContent(file);

08     //if the search pattern doesn't have any punctuation
09     //then assume it's not a regular expression and extract
10    //the entire line in which it was found
11     String[] regexTokens = searchPattern.split("\\p{Punct}");

12     if (regexTokens.length == 1)
13     {
14         searchPattern  = "^.*"+ searchPattern+".*$";
15     }

16    Map retval = searchString(fileContent,searchPattern,flags);
17    return retval;
18 }
```

Discussion Point

At this point, there should be some reasonable questions on your mind. Isn't this supposed to be a regex book? There wasn't anything particularly regex-like about the search file and search string methods; they were pretty much straight Java code, which you already know how to write. What's going on?

The point here is that regex is just a tool. It doesn't change the fact that you're still writing Java code, and you need to follow good, modular, object-oriented principles, even as you're working with regular expressions. Regex allows you to bridge trouble spots you might never have crossed otherwise, but it's just a tool. Like any well-built engine, the `java.util.regex` engine announces its excellence by humming quietly along and *not* forcing you to worry about it.

Working with Very Large Files

Another valid question at this point is, what if the content of the file you're trying to parse is too large to make reading all of it into memory a practical option? In general, you have two paths you can take here. You can use one of the new Java features, such as `MappedByteBuffer`s, or you can split the file into manageable sections and parse each of those in turn.

If you decide to use `MappedByteBuffer`s for regex, Listing 5-10 contains an example showing how. I'm hesitant, however, to advocate `MappedByteBuffer`s with regex too strongly for three reasons. First and foremost, their behavior is very system dependent, so you should probably rule them out if you need platform independence. Second, even within a given platform, their behavior isn't well defined. Thus, depending on what else you're doing with your operating system, you could get inconsistent results. Third, you need to consider the fact that, if the entire file can't be loaded into memory at one time, trying to apply a pattern that might have wildcards in it is going to be a tricky affair.

You may very well need to reconsider your patterns, and break the file up into logical blocks based on your insight into its structure. One strategy might be to check the size of the file and divide that by 10, 100, or whatever fraction is easily loadable given your system's memory limitations, and then search that portion. Although this isn't ideal, it is more predictable than the corresponding mapped-memory approach. The bottom line is that regardless of the regex flavor or provider you use, very large files require special treatment.

Listing 5-10. Accessing a File Through a MappedByteBuffer

```
01   public static boolean getFileContentUsingMappedByteBuffer
02   (
03       String fileName
04   ) throws IOException
05   {
06       boolean retval = false;
07       RandomAccessFile raf = new RandomAccessFile(fileName,"rwd");
08       FileChannel fc = raf.getChannel();

09       MappedByteBuffer mbb =
10        fc.map(FileChannel.MapMode.READ_WRITE,0,fc.size());

11      CharSequence cb = mbb.asCharBuffer();

12      return retval;
13   }
```

Modifying the Contents of a File

Now I want to provide a facility that modifies the content of a file based on a regex pattern. That is, I want to provide a mechanism that opens a file, searches its contents based on a regex pattern, and changes every occurrence of that pattern with a replacement string. Because I already have code that will open and search a file, modifying the content of the file is fairly easy. The logic for doing so is shown in Figure 5-2.

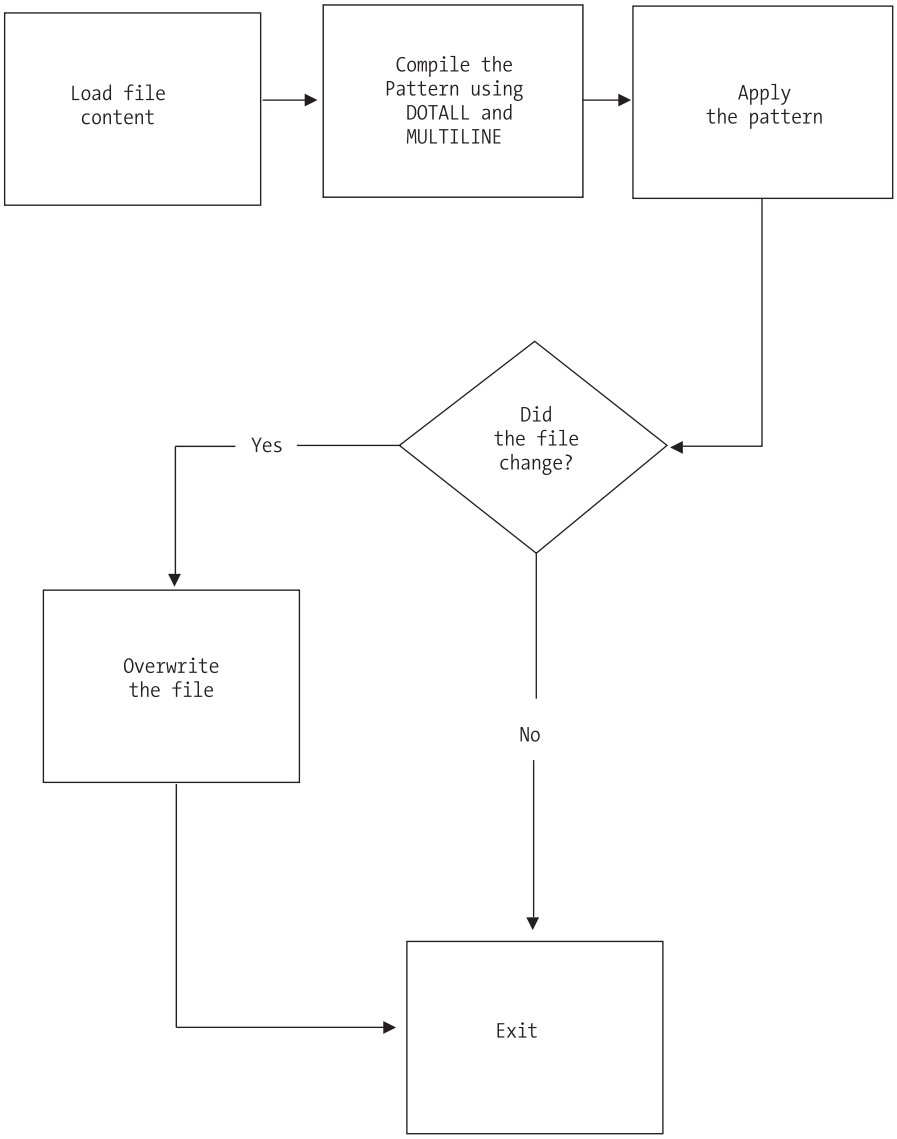

Figure 5-2. Basic flow diagram for updating the content of a file

Again, I decide to use a `FileChannel` for efficiency, as shown in Listing 5-11.

Listing 5-11. Modifying the Content of a File Based on a Regex Pattern

```
/**
* updates the content of the file. By default, the
* Pattern.MULTILINE is used. Also supports the
* $d notation in the replacement string, per the
* Matcher.replaceAll method
* @param the String fileName is the name and file path
* @param the String regex pattern to look for
* @param the String replacement for the regex
* @throws IOException if there is an IO error
*
* @return boolean true if the file was updated
*/
public static boolean updateFileContent
(
   String fileName,
   String regex,
   String replacement
) throws IOException
{
   boolean retval = false;

   RandomAccessFile raf =
       new RandomAccessFile(fileName,"rwd");
   FileChannel fc = raf.getChannel();

   String fileContent = getFileContent(fc);
   //Activate the MULTILINE flag for this regex
   regex = "(?m)"+regex;

   String newFileContent =
       fileContent.replaceAll(regex,replacement);

   //if nothing changed, then don't update the file
   if (!newFileContent.equals(fileContent))
   {
      setFileContent(newFileContent,fc);
      retval = true;
   }
```

```
35     //close up shop
36     fc.close();
37     fc = null;
38     raf = null;

39     return retval;
40  }

41  /**
42  * sets the content of a file. Completely
43  * overwrites previous file content, and truncates
44  * file to the length of the new content.
45  * @param the <code>String</code> newContent
46  * @param the <code>FileChannel</code> fc
47  * @throws <code>IOException</code>
48  *
49  * @author M Habibi
50  */
51  private static void setFileContent(
52     String newContent, FileChannel fc
53  )
54  throws IOException{
55     //write out the content to the file
56     ByteBuffer bb = ByteBuffer.wrap(newContent.getBytes());
57     //truncate the size of the file, in case the
58     //original file content was longer the new
59     //content
60     fc.truncate(newContent.length());

61     //start writing as position 0
62     fc.position(0);
63     fc.write(bb);

64     fc.close();
65     fc = null;
66  }
```

Listing 5-11 takes advantage of the `getFileContent` method defined earlier in line 29 of Listing 5-7. Otherwise, the example is self-contained.

Extracting Phone Numbers from a File

In this example, I want to parse a file and extract any and all phone numbers. This is a program I wrote to help a friend who owns a small IT shop. He had all sorts of electronic documents and needed to extract phone numbers from them to call his clients back. I'll start this process at the very beginning, where I extracted requirements:

Question: Are you looking for U.S. numbers or international ones?

Answer: U.S. numbers, but that could change.

Q: Is speed an issue? Is someone going to be tapping his foot, waiting for this to finish?

A: No, running overnight is fine.

Q: Do the phone numbers follow any sort of consistent format?

A: They're either seven or ten digits.

Q: Do they have hyphens or spaces in them?

A: It depends—sometimes they do.

Q: Is the format of the file subject to change?

A: Yes.

Q: If there had to be a mistake, would you prefer too many phone number candidates or too few?

A: Too many.

Q: Do you need these numbers returned in any particular kind of format?

A: I hadn't thought of that, but a consistent format would be great.

Q: Do you have some files you've already looked through that I can use for testing?

A: Yes.

Q: How big are these files?

A: Not that big. I don't know.

Q: How many files are there?

A: About ten per night.

Q: What types of files are these?

A: Microsoft Word documents.

I think I have enough information at this point to get started. It sounds like the client wants anything that might be a seven- or ten-digit phone number, and that speed isn't an issue. It also sounds like the files don't get that large. This should be as simple as defining a phone number pattern and using the previously presented search methods. After all, I can already access a file and search its content. I decide to keep the actual regex in a external property file, of course, so I can tweak it as I need to. This is going to be an error-prone process until I get a sense of these files.

I'm ready to start. I decide to do a quick search of the Web, and I come up with a few patterns for phone numbers. Some of these are a little esoteric, but I'm willing to try them because my client wants as many candidates as possible. The patterns I found are as follows:

```
^(\(?\+?[0-9]*\)?)?[0-9_\- \(\)]*$
^([0-1]([\s-./\\])?)?(\(?[2-9]\d{2}\)?
[2-9]\d{3})([\s-./\\])?(\d{3}([\s-./\\])?\d{4}
[a-zA-Z0-9]{7})$
^\(?[\d]{3}\)?[\s-]?[\d]{3}[\s-]?[\d]{4}$
```

I write a quick pattern that ORs these patterns together, run through some documents, and find that, in fact, it doesn't work. Although I'm getting some phone numbers back, I'm also getting some numbers that can't possibly be phone number patterns because they include characters, long spaces, and punctuation.

I have two choices here: I can run a second validation on the candidates that do match, or I can tweak the pattern. This time, I decide to take my own pattern from earlier and try to work in the good traits from the other patterns, as shown in Table 5-5. This is the composition technique introduced in Chapter 1.

Table 5-5. Pulling a General Regex Pattern from 614-345-6789

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 1	Nothing	Initial state	N/A	(?:\d{3})?\d{3}\d{4}
Step 2	Put optional - between the number groups	To get a more generic description	The phone number can be delimited with punctuation.	(?:\d{3}-?)?\d{3}-?\d{4}
Step 3	Put optional spaces between the number groups	To get a more generic description	The phone number can be delimited with spaces.	(?:\d{3}-?\s?)?\d{3}-?\s?\d{4}

Table 5-5. Pulling a General Regex Pattern from 614-345-6789 (Continued)

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 4	Swapped out - for ***\p{Punct}***	To accommodate punctuation	***\p{Punct}*** is a superset of -	(?:\d{3}\p{Punct}?\s?)?\d{3}\p{Punct}?\s?\d{4}
Step 5	Replaced ***(?:\d{3}\p{Punct}?\s?)?\d{3}\p{Punct}?*** with ***(?\d{3}\p{Punct}?\s?){1,2}***	To create a more succinct pattern	The two are equivalent statements.	(?:\d{3}\p{Punct}?\s?){1,2}\d{4}

Using this pattern, I find that things seem rational. Finally, just before finishing, I decide to format all of my output in the form *ddd-dddd* or *ddd-ddd-dddd*. The resulting code is shown in Listing 5-12.

Listing 5-12. Extracting Phone Numbers from a File

```
01  /**
02  * mines phone numbers out of the given file, and returns
03  * them as strings.
04  * @param the String filePath of the file
05  * @throws IOException if the file is not found or
06  * is corrupted
07  *
08  * @return ArrayList contained well formatted phone numbers
09  * of the for ddd-ddd-dddd or ddd-dddd
10 */
11  public static ArrayList minePhoneNumbers(String filePath)
12  throws IOException{

13     ArrayList retval = new ArrayList();
14     //get pattern
15     String regex = RegexUtil.getProperty("../regex.properties","allPhones");
```

```
16     //find all the matches
17     Map result =
18        RegexUtil.searchFile(filePath, regex ,Pattern.MULTILINE);

19     //get the matching strings
20     Iterator it = result.values().iterator();

21     //provide a consistent format for phone numbers captured
22     while (it.hasNext())
23     {
24        String num = (String)it.next();
25        num = num.replaceAll("\\p{Punct}|\\s","");

26        if (num.length() == 7)
27          num=num.replaceAll("(\\d{3})(\\d{4})","$1-$2");
28        else
29         num=num.replaceAll("(\\d{3})(\\d{3})(\\d{4})","($1)-$2-$3");

30       retval.add(num);
31     }

32     return retval;
33  }
```

Listing 5-12 is fairly self-explanatory. However, I do want to point out lines 27 and 29. Notice how easy it was to make a minor adjustment and produce well-formatted, consistent output here. Line 27, for example, simply says, "I'd like to capture the first three digits in group number 1 and the last four digits in group number 2. Then, I'd like to separate those two groups with a hyphen." Again, this involved very easy, but ultimately very powerful, code.

Searching a Directory for a File That Contains a Regex Expression

In principle, searching a directory for a file that matches a particular pattern is fairly easy. I already have a mechanism in place to search a file, so all I have to do is search a list of files. As you can see in Figure 5-3, the algorithm is recursive.

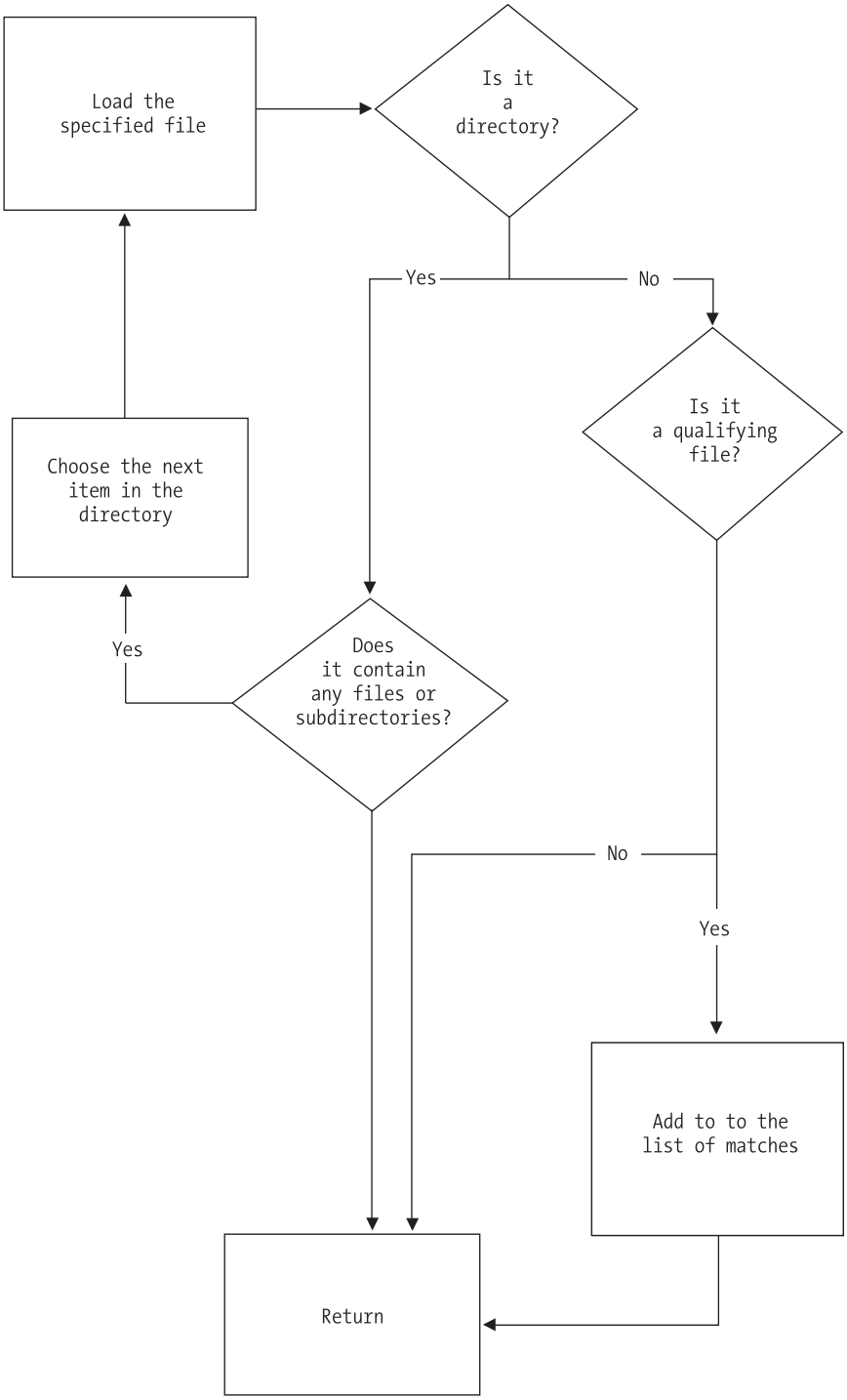

Figure 5-3. Recursive directory search

Listing 5-13 implements this design by taking advantage of the existing frameworks. It searches through subdirectories, looking for files that happen to contain the regex pattern I described.

Listing 5-13. Search Current and Subdirectories for File Containing the Pattern

```
01  /**
02  * Searches through the given directory and finds
03  * the specified files, based the searchPattern that describes
04  * its content. Returns matching files in an ArrayList. This
05  * method searches recursively through the file system.
06  * @param the File currentFile the directory, or file, to start
07  * searching in
08  * @param the String fileExtension, if any, of the file
09  * @param the String searchPattern, the regex that describes
10  * the file content we're looking for
11  * @param the int flags and flags we want to apply to the regex
12  * pattern.
13  * @throws IOException if there's an IO problem
14  *
15  * @return ArrayList containing <code>File</code> objects,
16  * or an empty ArrayList, if no matches are found
17  */
18  public static ArrayList searchDirs(
19      File currentFile,
20      String fileExtension,
21      String searchPattern,
22      int flags
23  ) throws IOException
24  {
25     ArrayList retval = new ArrayList();

26     if (!currentFile.isDirectory())
27     {
28         Map tmp = searchFile(
29             currentFile.getPath(),
30            searchPattern,flags);

31         //if anything was found, add the file
32         if (tmp.size() > 0)
33         {
34              retval.add(currentFile);
35              this.log.finest("added " + currentFile);
36         }
37     }
```

```
38      else
39      {   //step through subdirectories
40          File subs[]  =
41            currentFile.listFiles(
42                newLocalFileFilter(fileExtension));

43          if (subs != null)
44          {
45            //if the recursive search found anything, add it
46            for (int i=0; i < subs.length; i++)
47            {
48                ArrayList tmp=null;
49                tmp =searchDirs(
50                    subs[i],
51                    fileExtension,
52                    searchPattern,
53                    flags);

54                if (tmp.size() > 0)
55                {
56                   log.info(subs[i].getPath());
57                   retval.addAll(tmp);
58                }
59           }
60          }
61      }

62      return retval;
63   }
64  /**
65  * private filtering class, so that file
66  * searches can be more efficient
67  */
68  private static class LocalFileFilter implements FileFilter{
69      private String extension;
70      LocalFileFilter()
71      {
72          this(null);
73      }

74      LocalFileFilter(String extension)
75      {
76          this.extension = extension;
77      }
```

```
78      /**
79      * true if the current file meets the criteria
80      * @param the file pathname to check
81      *
82      * @return true if the file has the extension, or
83      * equals null, or the file is a directory.
84      * Else, returns false.
85      */
86      public boolean accept(File pathname){

87         boolean retval = false;
88         if (extension == null)
89         {
90             retval =  true;
91         }
92         else
93         {
94             String tmp = pathname.getPath();
95             if (tmp.endsWith(extension)) retval =  true;
96             if (pathname.isDirectory()) retval =  true;
97          }

98         return retval;
99      }
100 }
```

Validating an EDI Document

This next example is taken from a posting on the Sun site. A programmer needs help validating an Electronic Data Interchange (EDI) document. He needs to make sure that the `String` *ISA* always occurs before the `String` *IEA*, and that each occurs only once. He provided the sample input *ISA*XX*XXXXXXXXXXXXXXX*XX*XXXXXXXXXXXXXXX*030130*0912*~ IEA*1*000005900~*.

This problem is a candidate for the push technique, because it's fairly clear that I'll have to push the data into a pattern. To simplify the problem, I decide to deal in the abstract a bit. Instead of the strings *ISA* and *IEA*, I decide to use the @ sign and the # sign. Furthermore, I decide that everything—all the stuff in between @ and #—is a number. These are just logical placeholders, for my own benefit. I want to be able abstract away some of the messy details.

> **NOTE** If you happened to have liked mathematics in school, you'll notice that this is similar to the algebraic technique of factoring out messy subexpressions and referring to them using a simple variable.

Now I'll see if I can take this anywhere with the reasoning in Table 5-6.

Table 5-6. Pulling a General Regex Pattern from @45#78

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 1	Nothing	Initial state	N/A	@45#87
Step 2	Substituted *[^@]* for *4*	To get a more generic description	The only distinguishing feature of *4* is that it's not @, hence *[^@]*.	@[^@]5#7
Step 3	Substituted *[^@]* for *5*	To get a more generic description	The only distinguishing feature of *5* is that it's not @.	@[^@][^@]#7
Step 4	Swapped in *[^@]** for *[^@][^@]*	To get a more generic description	*[^@]** is a superset of *[^@]*.	@[^@]*#7
Step 5	Swapped in *([^@][^#])* for *7*	To get a more generic description	The only distinguishing feature of *7* is that it's not @ or *#*.	@[^@]*#([^@][^#])8
Step 6	Swapped in *([^@][^#])* for *8*	To get a more generic description	The only distinguishing feature of *8* is that it's not @ or *#*.	@[^@]*#([^@][^#])([^@][^#])
Step 7	Swapped in *([^@][^#])** for *([^@][^#])([^@][^#])*	To get a more generic description	*([^@][^#])** is a superset of *([^@][^#])([^@][^#])*.	@[^@]*#([^@][^#])*

I think I've taken that about as far as I can. Now I'll start stepping away from the abstract and heading back toward what I actually wanted. Table 5-7 breaks down my reasoning.

Table 5-7. Pulling an EDI Regex out of @[^@]#([^@][^#])**

Step	What I Did	Why I Did It	Justification	Resulting Pattern
Step 8	Nothing	Initial state	N/A	@[^@]*#([^@][^#])*
Step 9	Substitute *ISA* for @	To get a more specific description	@ was always just a stand-in for *ISA*.	ISA[^ISA]*#([^ISA][^#])*
Step 10	Substitute *IEA* for #	To get a more specific description	# was always just a stand-in for *IEA*.	ISA[^ISA]*IEA([^ISA][^IEA])*
Step 11	Added ***?:*** inside ***([^ISA][^IEA])***	To improve efficiency	I don't need a capturing group.	ISA[^ISA]*IEA(?:[^ISA][^IEA])*

Summary

The examples in this chapter should act as samples for writing your own regex expressions. I discussed confirming zip codes, using date formats, searching a string, searching a file, extracting data from a file, and modifying the contents of a file. If you're looking for more examples like these, please visit `http://www.infuxs.com`.

FAQs

Q: **Where can I get more information about a particular pattern?**

A: I suggest that you ask the folks at one of the various Java newsgroups. The JavaRanch site (`http://www.javaranch.com`) is particularly helpful, and so are the good folks on the ORO and Regexp newsgroups. Also, Apress now provides forums (`http://forums.apress.com`) where readers can interact directly with authors. When you ask your question, make sure you provide sample input, expected output, and your current code.

Q: **What are some other resources for regex?**

A: The single best regex book I can think of is Jeffery E. F. Friedl's *Mastering Regular Expressions* (O'Reilly & Associates, 2002). It's an elegant introduction to regex, and it deals with how several languages, including Java, use regex. It doesn't focus on Java regex with the detail that this book does, but it does provide an excellent, entertaining, and detailed account of regex mechanics, theory, and application. If you're looking for a book that will expand your general understanding of regex, you should seriously consider purchasing *Mastering Regular Expressions*.

Q: **Can I e-mail you with regex questions?**

A: Well, yes and no. I won't address private e-mail messages, but you're likely to find me lurking on the JavaRanch site (`http://www.javaranch.com`) and the Apress forums (`http://forums.apress.com`). If your questions are posted so that the general public can take advantage of the discussion, then I'll try and provide whatever help I can.