



[Java Cookbook, 2nd Edition](#)

By Ian F. Darwin

[Table of Contents](#)

---

## Chapter 4. Pattern Matching with Regular Expressions

[Introduction](#)

[Recipe 4.1. Regular Expression Syntax](#)

[Recipe 4.2. Using regexes in Java: Test for a Pattern](#)

[Recipe 4.3. Finding the Matching Text](#)

[Recipe 4.4. Replacing the Matched Text](#)

[Recipe 4.5. Printing All Occurrences of a Pattern](#)

[Recipe 4.6. Printing Lines Containing a Pattern](#)

[Recipe 4.7. Controlling Case in Regular Expressions](#)

[Recipe 4.8. Matching "Accented" or Composite Characters](#)

[Recipe 4.9. Matching Newlines in Text](#)

[Recipe 4.10. Program: Apache Logfile Parsing](#)

[Recipe 4.11. Program: Data Mining](#)

[Recipe 4.12. Program: Full Grep](#)

### Introduction

Suppose you have been on the Internet for a few years and have been very faithful about saving all your correspondence, just in case you (or your lawyers, or the prosecution) need a copy. The result is that you have a 50-megabyte disk partition dedicated to saved mail. And let's further suppose that you remember that somewhere in there is an email message from someone named Angie or Anjie. Or was it Angy? But you don't remember what you called it or where you stored it. Obviously, you have to look for it.

But while some of you go and try to open up all 15,000,000 documents in a word processor, I'll just find it with one simple command. Any system that provides regular expression support allows me to search for the pattern in several ways. The simplest to understand is:

```
Angie|Anjie|Angy
```

which you can probably guess means just to search for any of the variations. A more concise form ("more thinking, less typing") is:

```
An[ ^ dn]
```

to search in all the files. The syntax will become clear as we go through this chapter. Briefly, the "A" and the "n" match themselves, in effect finding words that begin with "An", while the cryptic [^ dn] requires the "An" to be followed by a character other than a space (to eliminate the very common English word "an" at the start of a sentence) or "d" (to eliminate the common word "and") or "n" (to eliminate Anne, Announcing, etc.). Has your word processor gotten past its splash screen yet? Well, it doesn't matter, because I've already found the missing file. To find the answer, I just typed the command:

```
grep 'An[ ^ dn]' *
```

*Regular expressions*, or regexes for short, provide a concise and precise specification of patterns to be matched in text.

As another example of the power of regular expressions, consider the problem of bulk-updating hundreds of files. When I started with Java, the syntax declaring array references was `baseType arrayVariableName[]`. For example, a method with an array argument, such as every program's main method, was commonly written as:

```
public static void main(String args[]) {
```

But as time went by, it became clear to the stewards of the Java language that it would be better to write it as `baseType[] arrayVariableName`, e.g.:

```
public static void main(String[] args) {
```

This is better Java style because it associates the "array-ness" of the type with the type itself, rather than with the local argument name. and the compiler now accents both modes. I wanted to

change all occurrences of `main` written the old way to the new way. I used the pattern `'main(String [a-z]'` with the `grep` utility described earlier to find the names of all the files containing old-style `main` declarations, that is, `main(String` followed by a space and a name character rather than an open square bracket. I then used another regex-based Unix tool, the stream editor `sed`, in a little shell script to change all occurrences in those files from `'main(String \([a-z][a-z]*\)'` to `'main(String[] \1'` (the syntax used here is discussed later in this chapter). Again, the regex-based approach was orders of magnitude faster than doing it interactively, even using a reasonably powerful editor such as `vi` or `emacs`, let alone trying to use a graphical word processor.

Unfortunately, the syntax of regexes has changed as they get incorporated into more tools<sup>[1]</sup> and more languages, so the exact syntax in the previous examples is not exactly what you'd use in Java, but it does convey the conciseness and power of the regex mechanism.

<sup>[1]</sup> Non-Unix fans fear not, for you can do this on Win32 using one of several packages. One is an open source package alternately called `CygWin` (after Cygnus Software) or `GnuWin32` (<http://sources.redhat.com/cygwin/>). Another is Microsoft's own `Unix Services for Windows`. Or you can use my `Grep` program in [Recipe 4.6](#) if you don't have `grep` on your system. Incidentally, the name `grep` comes from an ancient Unix line editor command `g/RE/p`, the command to find the regex globally in all lines in the edit buffer and print the lines that match—just what the `grep` program does to lines in files.

As a third example, consider parsing an Apache web server log file, where some fields are delimited with quotes, others with square brackets, and others with spaces. Writing ad-hoc code to parse this is messy in any language, but a well-crafted regex can break the line into all its constituent fields in one operation (this example is developed in [Recipe 4.10](#)).

These same time gains can be had by Java developers. Prior to 1.4, Java did not include any facilities for describing regular expressions in text. This is mildly surprising given how powerful regular expressions are, how ubiquitous they are on the Unix operating system (where Java was first brewed), and how powerful they are in modern scripting languages like `sed`, `awk`, `Python`, and `Perl`. [Table 4-1](#) lists about half a dozen regular expression packages for Java. I even wrote my own at one point; it worked well enough but was too slow for production use. The Apache Jakarta Regular Expressions and `ORO` packages are widely used.

**Table 4-1. Some Java regex packages**

Package	Notes	URL
JDK 1.4 API	Package <code>java.util.regex</code>	<a href="http://java.sun.com/">http://java.sun.com/</a>
Richard Emberson's	Unknown license; not being maintained	None; posted to <code>advanced-java@berkeley.edu</code> in 1998

Apache Jakarta ORO(original by Daniel Savarese)	Apache license; more comprehensive than Jakarta RegExp	<a href="http://jakarta.apache.org/oro/">http://jakarta.apache.org/oro/</a>
GNU Java Regexp	Lesser GNU Public License	<a href="http://www.cacas.org/java/gnu/regexp/">http://www.cacas.org/java/gnu/regexp/</a>

With JDK 1.4 and later, regular expression support is built into the standard Java runtime. The advantage of using the JDK 1.4 package is its integration with the runtime, including the standard class `java.lang.String` and the "new I/O" package. In addition to this integration, the JDK 1.4 package is one of the fastest Java implementations. However, code using any of the other packages still works, and you will find existing applications using some of these packages for the next few years since the syntax of each package is slightly different and it's not necessary to convert. Any new development, though, should be based on the JDK 1.4 regex package.

The first edition of this book focused on the Jakarta RegExp package; this edition covers the JDK 1.4 Regular Expressions API and does not cover any other package. The syntax of regexes themselves is discussed in [Recipe 4.1](#), and the syntax of the Java API for using regexes is described in [Recipe 4.2](#). The remaining recipes show some applications of regex technology in JDK 1.4.

### See Also

Mastering Regular Expressions by Jeffrey E. F. Friedl (O'Reilly), now in its second edition, is the definitive guide to all the details of regular expressions. Most introductory books on Unix and Perl include some discussion of regexes; Unix Power Tools (O'Reilly) devotes a chapter to them.

## Recipe 4.1. Regular Expression Syntax

### Problem

You need to learn the syntax of JDK 1.4 regular expressions.

### Solution

Consult [Table 4-2](#) for a list of the regular expression characters.

### Discussion

These pattern characters let you specify regexes of considerable power. In building patterns, you can use any combination of ordinary text and the metacharacters, or special characters, in [Table 4-2](#). These can all be used in any combination that makes sense. For example, `a+` means any number of occurrences of the letter `a`, from one up to a million or a gazillion. The pattern `Mrs?\.` matches `Mr.` or `Mrs.`. And `.*` means "any character, any number of times," and is similar in meaning to most command-line interpreters' meaning of the `*` alone. The pattern `\d+` means any

number of numeric digits. `\d{2,3}` means a two- or three-digit number.

Table 4-2. Regular expression metacharacter syntax		
Subexpression	Matches	Notes
General		
<code>^</code>	Start of line/string	
<code>\$</code>	End of line/string	
<code>\b</code>	Word boundary	
<code>\B</code>	Not a word boundary	
<code>\A</code>	Beginning of entire string	
<code>\Z</code>	End of entire string	
<code>\Z</code>	End of entire string (except allowable	See <a href="#">Recipe 4.9</a>
<code>.</code>	Any one character (except line	
<code>[...]</code>	"Character class": any one character	
<code>[^...]</code>	Any one character not from those	See <a href="#">Recipe 4.2</a>
Alternation and grouping		
<code>(...)</code>	Grouping (capture groups)	See <a href="#">Recipe 4.3</a>
<code> </code>	Alternation	
<code>(?:re)</code>	Noncapturing parenthesis	
<code>\G</code>	End of the previous match	
<code>\n</code>	Back-reference to capture group	
Normal (greedy) multipliers		
<code>{m,n}</code>	Multiplier for "from <i>m</i> to <i>n</i> repetitions"	See <a href="#">Recipe 4.4</a>
<code>{m,}</code>	Multiplier for " <i>m</i> or more repetitions"	
<code>{m}</code>	Multiplier for "exactly <i>m</i> repetitions"	See <a href="#">Recipe 4.10</a>
<code>{,n}</code>	Multiplier for 0 up to <i>n</i> repetitions	
<code>*</code>	Multiplier for 0 or more repetitions	Short for <code>{0,}</code>
<code>+</code>	Multiplier for 1 or more repetitions	Short for <code>{1,}</code> ; see <a href="#">Recipe 4.2</a>
<code>?</code>	Multiplier for 0 or 1 repetitions (i.e.	Short for <code>{0,1}</code>

	present exactly once, or not at all)	
Reluctant (non-greedy) multipliers		
$\{m,n\}?$	Reluctant multiplier for "from $m$ to $n$ repetitions"	
$\{m,\}$ ?	Reluctant multiplier for " $m$ or more repetitions"	
$\{,n\}?$	Reluctant multiplier for 0 up to $n$ repetitions	
$*?$	Reluctant multiplier: 0 or more	
$+?$	Reluctant multiplier: 1 or more	See <a href="#">Recipe 4.10</a>
$??$	Reluctant multiplier: 0 or 1 times	
Possessive (very greedy) multipliers		
$\{m,n\}+$	Possessive multiplier for "from $m$ to $n$ repetitions"	
$\{m,\}+$	Possessive multiplier for " $m$ or more repetitions"	
$\{,n\}+$	Possessive multiplier for 0 up to $n$ repetitions	
$*+$	Possessive multiplier: 0 or more	
$++$	Possessive multiplier: 1 or more	
$?+$	Possessive multiplier: 0 or 1 times	
Escapes and shorthands		
$\$	Escape (quote) character: turns most metacharacters off; turns subsequent alphabetic into metacharacters	
$\Q$	Escape (quote) all characters up to $\E$	
$\E$	Ends quoting begun with $\Q$	
$\t$	Tab character	
$\r$	Return (carriage return) character	
$\n$	Newline character	See <a href="#">Recipe 4.9</a>
$\f$	Form feed	
$\w$	Character in a word	Use $\w+$ for a word; see <a href="#">Recipe 4.10</a>
$\W$	A non-word character	
$\d$	Numeric digit	Use $\d+$ for an integer; see <a href="#">Recipe 4.9</a>

<code>\D</code>	A non-digit character	
<code>\s</code>	Whitespace	Space, tab, etc., as determined by <code>java.lang.Character.isWhitespace()</code>
<code>\S</code>	A nonwhitespace character	See <a href="#">Recipe 4.10</a>
Unicode blocks (representative samples)		
<code>\p{InGreek}</code>	A character in the Greek block	(simple block)
<code>\P{InGreek}</code>	Any character not in the Greek block	
<code>\p{Lu}</code>	An uppercase letter	(simple category)
<code>\p{Sc}</code>	A currency symbol	
POSIX-style character classes (defined only for US-ASCII)		
<code>\p{Alnum}</code>	Alphanumeric characters	[A-Za-z0-9]
<code>\p{Alpha}</code>	Alphabetic characters	[A-Za-z]
<code>\p{ASCII}</code>	Any ASCII character	[\x00-\x7F]
<code>\p{Blank}</code>	Space and tab characters	
<code>\p{Space}</code>	Space characters	[ \t\n\x0B\f\r]
<code>\p{Cntrl}</code>	Control characters	[\x00-\x1F\x7F]
<code>\p{Digit}</code>	Numeric digit characters	[0-9]
<code>\p{Graph}</code>	Printable and visible characters (not spaces or control characters)	
<code>\p{Print}</code>	Printable characters	Same as <code>\p{Graph}</code>
<code>\p{Punct}</code>	Punctuation characters	One of !"#\$%&'()*+,-./:;<=>?@[\\]^_`{ }~
<code>\p{Lower}</code>	Lowercase characters	[a-z]
<code>\p{Upper}</code>	Uppercase characters	[A-Z]
<code>\p{XDigit}</code>	Hexadecimal digit characters	[0-9a-fA-F]

Regexes match anyplace possible in the string. Patterns followed by a *greedy* multiplier (the only type that existed in traditional Unix regexes) consume (match) as much as possible without compromising any subexpressions which follow; patterns followed by a *possessive* multiplier match as much as possible without regard to following subexpressions; patterns followed by a *reluctant* multiplier consume as few characters as possible to still get a match.

Also, unlike regex packages in some other languages, the JDK 1.4 package was designed to handle Unicode characters from the beginning. And the standard Java escape sequence `\unmmn` is used to specify a Unicode character in the pattern. We use methods of `java.lang.Character` to determine Unicode character properties, such as whether a given character is a space.

To help you learn how regexes work, I provide a little program called REDemo.<sup>[2]</sup> In the online directory *javasrc/RE*, you should be able to type either `ant REDemo`, or `javac REDemo` followed by `java REDemo`, to get the program running.

<sup>[2]</sup> REDemo was inspired by (but does not use any code from) a similar program provided with the Jakarta Regular Expressions package mentioned in the Introduction to [Chapter 4](#).

In the uppermost text box (see [Figure 4-1](#)), type the regex pattern you want to test. Note that as you type each character, the regex is checked for syntax; if the syntax is OK, you see a checkmark beside it. You can then select Match, Find, or Find All. Match means that the entire string must match the regex, while Find means the regex must be found somewhere in the string (Find All counts the number of occurrences that are found). Below that, you type a string that the regex is to match against. Experiment to your heart's content. When you have the regex the way you want it, you can paste it into your Java program. You'll need to escape (backslash) any characters that are treated specially by both the Java compiler and the JDK 1.4 regex package, such as the backslash itself, double quotes, and others (see the sidebar [Remember This!](#)).

## Remember This!

Remember that because a regex compiles strings that are also compiled by `javac`, you usually need two levels of escaping for any special characters, including backslash, double quotes, and so on. For example, the regex:

```
"You said it\."
```

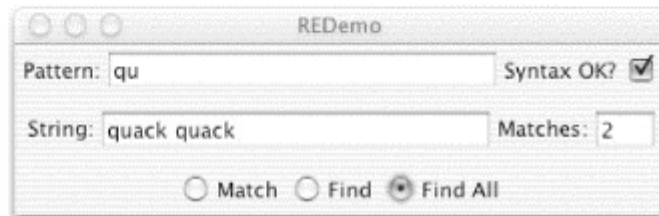
has to be typed like this to be a Java language String:

```
"\"You said it\\.\""
```

I can't tell you how many times I've made the mistake of forgetting the extra backslash in `\d+`, `\w+`, and their kin!

In [Figure 4-1](#), I typed `qu` into the REDemo program's Pattern box, which is a syntactically valid regex pattern: any ordinary characters stand as regexes for themselves, so this looks for the letter `q` followed by `u`. In the top version, I typed only a `q` into the string, which is not matched. In the second, I have typed `quack` and the `q` of a second `quack`. Since I have selected Find All, the count shows one match. As soon as I type the second `u`, the count is updated to two, as shown in the third version.

**Figure 4-1. REDemo with simple examples**



Regexes can do far more than just character matching. For example, the two-character regex `^T` would match beginning of line (`^`) immediately followed by a capital `T`—i.e., any line beginning with a capital `T`. It doesn't matter whether the line begins with `Tiny trumpets`, `Titanic tubas`, or `Triumphant slide trombones`, as long as the capital `T` is present in the first position.

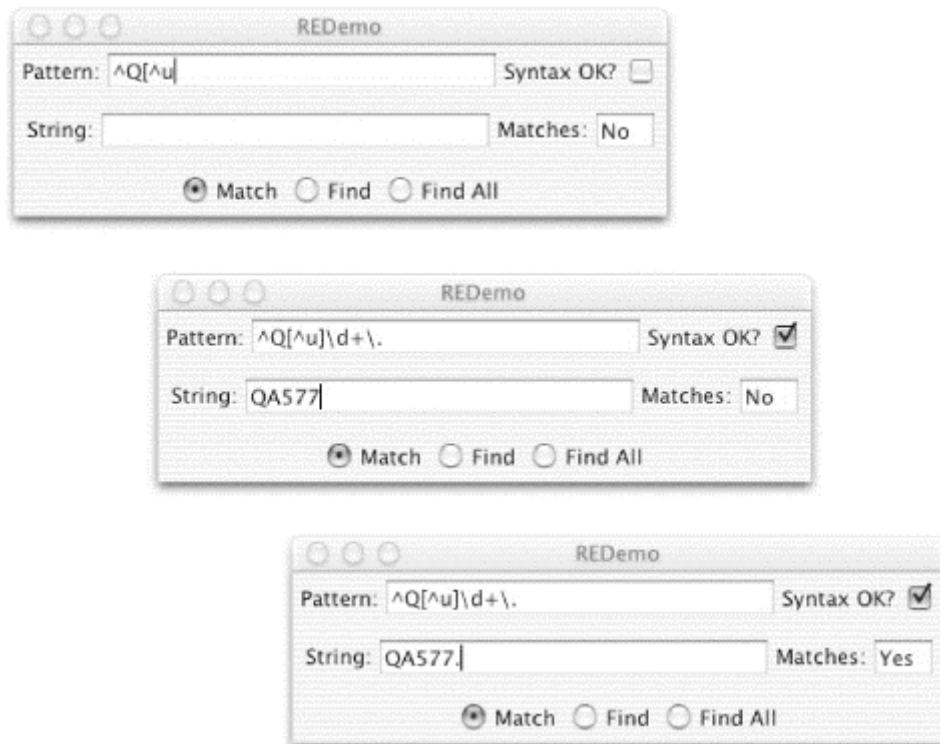
But here we're not very far ahead. Have we really invested all this effort in regex technology just to be able to do what we could already do with the `java.lang.String` method `startsWith()`? Hmm, I can hear some of you getting a bit restless. Stay in your seats! What if you wanted to match not only a letter `T` in the first position, but also a vowel (`a`, `e`, `i`, `o`, or `u`) immediately after it, followed by any number of letters in a word, followed by an exclamation point? Surely you could do this in Java by checking `startsWith("T")` and `charAt(1) == 'a' || charAt(1) == 'e'`, and so on? Yes, but by the time you did that, you'd have written a lot of very highly specialized code that you couldn't use in any other application. With regular expressions, you can just give the pattern `^T[aeiou]\w*!`. That is, `^` and `T` as before, followed by a *character class* listing the vowels, followed by any number of word characters (`\w*`), followed by the exclamation point.

"But wait, there's more!" as my late, great boss Yuri Rubinsky used to say. What if you want to be able to change the pattern you're looking for at runtime? Remember all that Java code you just wrote to match `T` in column 1, plus a vowel, some word characters, and an exclamation point? Well, it's time to throw it out. Because this morning we need to match `Q`, followed by a letter other than `u`, followed by a number of digits, followed by a period. While some of you start writing a new function to do that, the rest of us will just saunter over to the RegEx Bar & Grille, order a `^Q[^u]\d+.` from the bartender, and be on our way.

OK. the `[^u]` means "match any one character that is not the character `u`." The `\d+` means one or

more numeric digits. The + is a *multiplier* or *quantifier* meaning one or more occurrences of what it follows, and \d is any one numeric digit. So \d+ means a number with one, two, or more digits. Finally, the \. Well, . by itself is a metacharacter. Most single metacharacters are switched off by preceding them with an escape character. Not the ESC key on your keyboard, of course. The regex "escape" character is the backslash. Preceding a metacharacter like . with escape turns off its special meaning. Preceding a few selected alphabetic characters (e.g., n, r, t, s, w) with escape turns them into metacharacters. [Figure 4-2](#) shows the ^Q[^\u]\d+\. regex in action. In the first frame, I have typed part of the regex as ^Q[^\u and, since there is an unclosed square bracket, the Syntax OK flag is turned off; when I complete the regex, it will be turned back on. In the second frame, I have finished the regex and typed the string as QA577 (which you should expect to match the ^Q[^\u]\d+, but not the period since I haven't typed it). In the third frame, I've typed the period so the Matches flag is set to Yes.

**Figure 4-2. REDemo with ^Q[^\u]\d+\. example**



One good way to think of regular expressions is as a "little language" for matching patterns of characters in text contained in strings. Give yourself extra points if you've already recognized this as the design pattern known as *Interpreter*. A regular expression API is an interpreter for matching regular expressions.

So now you should have at least a basic grasp of how regexes work in practice. The rest of this chapter gives more examples and explains some of the more powerful topics, such as capture groups. As for how regexes work in theory—and there is a lot of theoretical details and differences among regex flavors—the interested reader is referred to the book *Mastering Regular Expressions*. Meanwhile, let's start learning how to write Java programs that use regular

expressions.

## Recipe 4.2. Using regexes in Java: Test for a Pattern

### Problem

You're ready to get started using regular expression processing to beef up your Java code by testing to see if a given pattern can match in a given string.

### Solution

Use the Java Regular Expressions Package, `java.util.regex`.

### Discussion

The good news is that the Java API for regexes is actually easy to use. If all you need is to find out whether a given regex matches a string, you can use the convenient boolean `matches()` method of the `String` class, which accepts a regex pattern in `String` form as its argument:

```
if (inputString.matches(stringRegexPattern)) {
    // it matched... do something with it...
}
```

This is, however, a convenience routine, and convenience always comes at a price. If the regex is going to be used more than once or twice in a program, it is more efficient to construct and use a `Pattern` and its `Matcher(s)`. A complete program constructing a `Pattern` and using it to match is shown here:

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

/**
 * Simple example of using regex class.
 */
public class RESimple {
    public static void main(String[] argv) throws PatternSyntaxException {
        String pattern = "^Q[^u]\\d+\\. ";
        String input = "QA777. is the next flight. It is on time.";

        Pattern p = Pattern.compile(pattern);

        boolean found = p.matcher(input).lookingAt( );

        System.out.println("'" + pattern + "'" +
            (found ? " matches '" : " doesn't match '" ) + input + "'");
    }
}
```

The `java.util.regex` package consists of two classes, `Pattern` and `Matcher`, which provide the public API shown in [Example 4-1](#).

### Example 4-1. Regex public API

```
/** The main public API of the java.util.regex package.
 * Prepared by javap and Ian Darwin.
 */

package java.util.regex;

public final class Pattern {
    // Flags values ('or' together)
    public static final int
        UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE,
        DOTALL, UNICODE_CASE, CANON_EQ;
    // Factory methods (no public constructors)
    public static Pattern compile(String patt);
    public static Pattern compile(String patt, int flags);
    // Method to get a Matcher for this Pattern
    public Matcher matcher(CharSequence input);
    // Information methods
    public String pattern( );
    public int flags( );
    // Convenience methods
    public static boolean matches(String pattern, CharSequence input);
    public String[] split(CharSequence input);
    public String[] split(CharSequence input, int max);
}

public final class Matcher {
    // Action: find or match methods
    public boolean matches( );
    public boolean find( );
    public boolean find(int start);
    public boolean lookingAt( );
    // "Information about the previous match" methods
    public int start( );
    public int start(int whichGroup);
    public int end( );
    public int end(int whichGroup);
    public int groupCount( );
    public String group( );
    public String group(int whichGroup);
    // Reset methods
    public Matcher reset( );
    public Matcher reset(CharSequence newInput);
    // Replacement methods
    public Matcher appendReplacement(StringBuffer where, String newText);
    public StringBuffer appendTail(StringBuffer where);
    public String replaceAll(String newText);
    public String replaceFirst(String newText);
    // information methods
    public Pattern pattern( );
}

/* String, showing only the regex-related methods */
public final class String {
    public boolean matches(String regex);
    public String replaceFirst(String regex, String newStr);
    public String replaceAll(String regex, String newStr)
    public String[] split(String regex)
    public String[] split(String regex, int max);
}
```

This API is large enough to require some explanation. The normal steps for regex matching in a production program are:

1. Create a Pattern by calling the static method `Pattern.compile()` .
2. Request a Matcher from the pattern by calling `pattern.matcher(CharSequence)` for each String (or other `CharSequence`) you wish to look through.
3. Call (once or more) one of the finder methods (discussed later in this section) in the resulting `Matcher`.

The `CharSequence` interface, added to `java.lang` with JDK 1.4, provides simple read-only access to objects containing a collection of characters. The standard implementations are `String` and `StringBuffer` (described in [Chapter 3](#)), and the "new I/O" class `java.nio.CharBuffer`.

Of course, you can perform regex matching in other ways, such as using the convenience methods in `Pattern` or even in `java.lang.String`. For example:

```
// StringConvenience.java -- show String convenience routine for "match"
String pattern = ".*Q[^u]\\d+\\.\\..*";
String line = "Order QT300. Now!";
if (line.matches(pattern)) {
    System.out.println(line + " matches \"" + pattern + "\"");
} else {
    System.out.println("NO MATCH");
}
```

But the three-step list just described is the "standard" pattern for matching. You'd likely use the `String` convenience routine in a program that only used the regex once; if the regex were being used more than once, it is worth taking the time to "compile" it, since the compiled version runs faster.

As well, the `Matcher` has several finder methods, which provide more flexibility than the `String` convenience routine `match()`. The `Matcher` methods are:

`match()`

Used to compare the entire string against the pattern; this is the same as the routine in `java.lang.String`. Since it matches the entire `String`, I had to put `.*` before and after the pattern.

`lookingAt()`

Used to match the pattern only at the beginning of the string.

`find()`

Used to match the pattern in the string (not necessarily at the first character of the string), starting at the beginning of the string or, if the method was previously called and succeeded, at the first character not matched by the previous match.

Each of these methods returns `boolean`, with `true` meaning a match and `false` meaning no match. To check whether a given string matches a given pattern, you need only type something like the

following:

```
Matcher m = Pattern.compile(patt).matcher(line);
if (m.find( )) {
    System.out.println(line + " matches " + patt)
}
```

But you may also want to extract the text that matched, which is the subject of the next recipe.

The following recipes cover uses of this API. Initially, the examples just use arguments of type `String` as the input source. Use of other `CharSequence` types is covered in Recipe [Recipe 4.5](#).

### Recipe 4.3. Finding the Matching Text

#### Problem

You need to find the text that the regex matched.

#### Solution

Sometimes you need to know more than just whether a regex matched a string. In editors and many other tools, you want to know exactly what characters were matched. Remember that with multipliers such as `*`, the length of the text that was matched may have no relationship to the length of the pattern that matched it. Do not underestimate the mighty `*`, which happily matches thousands or millions of characters if allowed to. As you saw in the previous recipe, you can find out whether a given match succeeds just by using `find()` or `matches()`. But in other applications, you will want to get the characters that the pattern matched.

After a successful call to one of the above methods, you can use these "information" methods to get information on the match:

`start()`, `end()`

Returns the character position in the string of the starting and ending characters that matched.

`groupCount()`

Returns the number of parenthesized capture groups if any; returns 0 if no groups were used.

`group(int i)`

Returns the characters matched by group *i* of the current match, if *i* is less than or equal to the return value of `groupCount()`. Group is the entire match, so `group(0)` (or just `group()`) returns the entire portion of the string that matched.

The notion of parentheses or "capture groups" is central to regex processing. Regexes may be nested to any level of complexity. The `group(int)` method lets you retrieve the characters that matched a given parenthesis group. If you haven't used any explicit parens, you can just treat whatever matched as "level zero." For example:

```
// Part of REmatch.java
String patt = "Q[^u]\\d+\\. ";
Pattern r = Pattern.compile(patt);
String line = "Order QT300. Now!";
Matcher m = r.matcher(line);
if (m.find( )) {
    System.out.println(patt + " matches \"" +
        m.group(0) +
        "\" in \"" + line + "\"");
} else {
    System.out.println("NO MATCH");
}
```

When run, this prints:

```
Q[^u]\\d+\\. matches "QT300." in "Order QT300. Now!"
```

An extended version of the REDemo program presented in [Recipe 4.2](#), called REDemo2, provides a display of all the capture groups in a given regex; one example is shown in [Figure 4-3](#).

**Figure 4-3. REDemo2 in action**



It is also possible to get the starting and ending indexes and the length of the text that the pattern matched (remember that terms with multipliers, such as the `\d+` in this example, can match an arbitrary number of characters in the string). You can use these in conjunction with the `String.substring()` methods as follows:

```
// Part of regexsubstr.java -- Prints exactly the same as REmatch.java
Pattern r = Pattern.compile(patt);
String line = "Order QT300. Now!";
Matcher m = r.matcher(line);
```

```

if (m.find( )) {
    System.out.println(patt + " matches \"" +
        line.substring(m.start(0), m.end(0)) +
        "\" in \"" + line + "\"");
} else {
    System.out.println("NO MATCH");
}
}

```

Suppose you need to extract several items from a string. If the input is:

```

Smith, John
Adams, John Quincy

```

and you want to get out:

```

John Smith
John Quincy Adams

```

just use:

```

// from REmatchTwoFields.java
// Construct a regex with parens to "grab" both field1 and field2
Pattern r = Pattern.compile("(.*), (.*?)");
Matcher m = r.matcher(inputLine);
if (!m.matches( ))
    throw new IllegalArgumentException("Bad input: " + inputLine);
System.out.println(m.group(2) + ' ' + m.group(1));

```

## Recipe 4.4. Replacing the Matched Text

As we saw in the previous recipe, regex patterns involving multipliers can match a lot of characters with very few metacharacters. We need a way to replace the text that the regex matched without changing other text before or after it. We could do this manually using the `String` method `substring( )`. However, because it's such a common requirement, the JDK 1.4 Regular Expression API provides some substitution methods. In all these methods, you pass in the replacement text or "right-hand side" of the substitution (this term is historical: in a command-line text editor's substitute command, the left-hand side is the pattern and the right-hand side is the replacement text). The replacement methods are:

`replaceAll(newString)`

Replaces all occurrences that matched with the new string.

`appendReplacement(StringBuffer, newString)`

Copies up to before the first match, plus the given newString.

appendTail(StringBuffer)

Appends text after the last match (normally used after appendReplacement).

[Example 4-2](#) shows use of these three methods.

#### **Example 4-2. ReplaceDemo.java**

```
// class ReplaceDemo
// Quick demo of substitution: correct "demon" and other
// spelling variants to the correct, non-satanic "daemon".

// Make a regex pattern to match almost any form (deamon, demon, etc.).
String patt = "d[ae]{1,2}mon"; // i.e., 1 or 2 'a' or 'e' any combo

// A test string.
String input = "Unix hath demons and deamons in it!";
System.out.println("Input: " + input);

// Run it from a regex instance and see that it works
Pattern r = Pattern.compile(patt);
Matcher m = r.matcher(input);
System.out.println("ReplaceAll: " + m.replaceAll("daemon"));

// Show the appendReplacement method
m.reset( );
StringBuffer sb = new StringBuffer( );
System.out.print("Append methods: ");
while (m.find( )) {
    m.appendReplacement(sb, "daemon"); // Copy to before first match,
    // plus the word "daemon"
}
m.appendTail(sb); // copy remainder
System.out.println(sb.toString( ));
```

Sure enough, when you run it, it does what we expect:

```
Input: Unix hath demons and deamons in it!
ReplaceAll: Unix hath daemons and daemons in it!
Append methods: Unix hath daemons and daemons in it!
```

## **Recipe 4.5. Printing All Occurrences of a Pattern**

### **Problem**

You need to find all the strings that match a given regex in one or more files or other sources.

## Solution

This example reads through a file one line at a time. Whenever a match is found, I extract it from the line and print it.

This code takes the `group()` methods from [Recipe 4.3](#), the `substring` method from the `CharacterIterator` interface, and the `match()` method from the `regex` and simply puts them all together. I coded it to extract all the "names" from a given file; in running the program through itself, it prints the words "import", "java", "until", "regex", and so on:

```
> jikes +E -d . ReaderIter.java
> java ReaderIter ReaderIter.java
import
java
util
regex
import
java
io
Print
all
the
strings
that
match
given
pattern
from
file
public
```

I interrupted it here to save paper. This can be written two ways, a traditional "line at a time" pattern shown in [Example 4-3](#) and a more compact form using "new I/O" shown in [Example 4-4](#) (the "new I/O" package is described in [Chapter 10](#)).

### Example 4-3. ReaderIter.java

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

/**
 * Print all the strings that match a given pattern from a file.
 */
public class ReaderIter {
    public static void main(String[] args) throws IOException {
        // The regex pattern
        Pattern patt = Pattern.compile("[A-Za-z][a-z]+");
        // A FileReader (see the I/O chapter)
        BufferedReader r = new BufferedReader(new FileReader(args[0]));

        // For each line of input, try matching in it.
        String line;
        while ((line = r.readLine( )) != null) {
            // For each match in the line, extract and print it.
            Matcher m = patt.matcher(line);
            while (m.find( )) {
                // Simplest method:
                // System.out.println(m.group(0));

                // Get the starting position of the text
                int start = m.start(0);
```

```

        // Get ending position
        int end = m.end(0);
        // Print whatever matched.
        System.out.println("start=" + start + "; end=" + end);
        // Use CharSequence.substring(offset, end);
        System.out.println(line.substring(start, end));
    }
}
}
}

```

#### Example 4-4. GrepNIO.java

```

import java.io.*;

import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.regex.*;

/* Grep-like program using NIO, but NOT LINE BASED.
 * Pattern and file name(s) must be on command line.
 */
public class GrepNIO {
    public static void main(String[] args) throws IOException {

        if (args.length < 2) {
            System.err.println("Usage: GrepNIO patt file [...]");
            System.exit(1);
        }

        Pattern p = Pattern.compile(args[0]);
        for (int i=1; i<args.length; i++)
            process(p, args[i]);
    }

    static void process(Pattern pattern, String fileName) throws IOException {

        // Get a FileChannel from the given file.
        FileChannel fc = new FileInputStream(fileName).getChannel( );

        // Map the file's content
        ByteBuffer buf = fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size( ));

        // Decode ByteBuffer into CharBuffer
        CharBuffer cbuf =
            Charset.forName("ISO-8859-1").newDecoder( ).decode(buf);

        Matcher m = pattern.matcher(cbuf);
        while (m.find( )) {
            System.out.println(m.group(0));
        }
    }
}

```

The NIO version shown in [Example 4-4](#) relies on the fact that an NIO Buffer can be used as a CharSequence. This program is more general in that the pattern argument is taken from the command-line argument. It prints the same output as the previous example if invoked with the pattern argument from the previous program on the command line:

```
java GrepNIO " [A-Za-z][a-z]+" ReaderIter.java
```

You might think of using `\w+` as the pattern; the only difference is that my pattern looks for well-formed capitalized words while `\w+` would include Java-centric oddities like `theVariableName`, which have capitals in nonstandard positions.

Also note that the NIO version will probably be more efficient since it doesn't reset the `Matcher` to a new input source on each line of input as `ReaderIter` does.

## **Recipe 4.6. Printing Lines Containing a Pattern**

### **Problem**

You need to look for lines matching a given regex in one or more files.

### **Solution**

Write a simple `grep`-like program.

### **Discussion**

As I've mentioned, once you have a regex package, you can write a `grep`-like program. I gave an example of the Unix `grep` program earlier. `grep` is called with some optional arguments, followed by one required regular expression pattern, followed by an arbitrary number of filenames. It prints any line that contains the pattern, differing from [Recipe 4.5](#), which prints only the

```

/** Grep0 - Match lines from stdin against the pattern on the command line.
 */
public class Grep0 {
    public static void main(String[] args) throws IOException {
        BufferedReader is =
            new BufferedReader(new InputStreamReader(System.in));
        if (args.length != 1) {
            System.err.println("Usage: Grep0 pattern");
            System.exit(1);
        }
        Pattern patt = Pattern.compile(args[0]);
        Matcher matcher = patt.matcher("");
        String line = null;
        while ((line = is.readLine( )) != null) {
            matcher.reset(line);
            if (matcher.find( )) {
                System.out.println("MATCH: " + line);
            }
        }
    }
}

```

## Recipe 4.7. Controlling Case in Regular Expressions

### Problem

You want to find text regardless of case.

### Solution

Compile the Pattern passing in the flags argument `Pattern.CASE_INSENSITIVE` to indicate that matching should be case-independent ("fold" or ignore differences in case). If your code might run in different locales (see [Chapter 15](#)), add `Pattern.UNICODE_CASE`. Without these flags, the default is normal, case-sensitive matching behavior. This flag (and others) are passed to the `Pattern.compile()` method, as in:

```

// CaseMatch.java
Pattern reCaseInsens = Pattern.compile(pattern, Pattern.CASE_INSENSITIVE |
Pattern.UNICODE_CASE);
reCaseInsens.matches(input);           // will match case-insensitively

```

This flag must be passed when you create the Pattern; as Pattern objects are immutable, they cannot be changed once constructed.

The full source code for this example is online as *CaseMatch.java*.

### Pattern.compile( ) Flags

Half a dozen flags can be passed as the second argument to `Pattern.compile()`. If more

than one value is needed, they can be or'd together using the | bitwise or operator. In alphabetical order, the flags are:

#### CANON\_EQ

Enables so-called "canonical equivalence," that is, characters are matched by their base character, so that the character e followed by the "combining character mark" for the acute accent ( ´ ) can be matched either by the composite character é or the letter e followed by the character mark for the accent (see [Recipe 4.8](#)).

#### CASE\_INSENSITIVE

Turns on case-insensitive matching (see Recipe [Recipe 4.7](#)).

#### COMMENTS

Causes whitespace and comments (from # to end-of-line) to be ignored in the pattern.

#### DOTALL

Allows dot (.) to match any regular character or the newline, not just newline (see Recipe [Recipe 4.9](#)).

#### MULTILINE

Specifies multiline mode (see Recipe [Recipe 4.9](#)).

#### UNICODE\_CASE

Enables Unicode-aware case folding (see [Recipe 4.7](#)).

#### UNIX\_LINES

Makes \n the only valid "newline" sequence for MULTILINE mode (see [Recipe 4.9](#)).

## Recipe 4.8. Matching "Accented" or Composite Characters

### Problem

You want characters to match regardless of the form in which they are entered.

### Solution

Compile the Pattern with the flags argument `Pattern.CANON_EQ` for "canonical equality."

### Discussion

Composite characters can be entered in various forms. Consider, as a single example, the letter `e` with an acute accent. This character may be found in various forms in Unicode text, such as the single character `é` (Unicode character `\u00e9`) or as the two-character sequence `e´` (`e` followed by the Unicode combining acute accent, `\u0301`). To allow you to match such characters regardless of which of possibly multiple "fully decomposed" forms are used to enter them, the `regex` package has an option for "canonical matching," which treats any of the forms as equivalent. This option is enabled by passing `CANON_EQ` as (one of) the flags in the second argument to `Pattern.compile()`. This program shows `CANON_EQ` being used to match several forms:

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

/**
 * CanonEqDemo - show use of Pattern.CANON_EQ, by comparing varous ways of
 * entering the Spanish word for "equal" and see if they are considered equal
 * by the regex-matching engine.
 */
public class CanonEqDemo {
    public static void main(String[] args) {
        String pattStr = "\u00e9gal"; // égal
        String[] input = {
            "\u00e9gal", // égal - this one had better match :-)
            "e\u0301gal", // e + "Combining acute accent"
            "e\u02c9gal", // e + "modifier letter acute accent"
            "e'gal", // e + single quote
            "e\u00b4gal", // e + Latin-1 "acute"
        };
        Pattern pattern = Pattern.compile(pattStr, Pattern.CANON_EQ);
        for (int i = 0; i < input.length; i++) {
            if (pattern.matcher(input[i]).matches( )) {
                System.out.println(pattStr + " matches input " + input[i]);
            } else {
                System.out.println(pattStr + " does not match input " + input[i]);
            }
        }
    }
}
```

When you run this program on JDK 1.4 or later, it correctly matches the "combining accent" and rejects the other characters, some of which, unfortunately, look like the accent on a printer, but are not considered "combining accent" characters.

```
égal matches input égal
égal matches input e?gal
égal does not match input e?gal
égal does not match input e'gal
```

égal does not match input e´ gal

For more details, see the character charts at <http://www.unicode.org/>.

## Recipe 4.9. Matching Newlines in Text

### Problem

You need to match newlines in text.

### Solution

Use `\n` or `\r`.

See also the flags constant `Pattern.MULTILINE`, which makes newlines match as beginning-of-line and end-of-line (`^` and `$`).

### Discussion

While line-oriented tools from Unix such as `sed` and `grep` match regular expressions one line at a time, not all tools do. The `sam` text editor from Bell Laboratories was the first interactive tool I know of to allow multiline regular expressions; the Perl scripting language followed shortly. In the Java API, the newline character by default has no special significance. The `BufferedReader` method `readLine()` normally strips out whichever newline characters it finds. If you read in gobs of characters using some method other than `readLine()`, you may have some number of `\n`, `\r`, or `\r\n` sequences in your text string.<sup>[4]</sup> Normally all of these are treated as equivalent to `\n`. If you want only `\n` to match, use the `UNIX_LINES` flag to the `Pattern.compile()` method.

<sup>[4]</sup> Or a few related Unicode characters, including the next-line (`\u0085`), line-separator (`\u2028`), and paragraph-separator (`\u2029`) characters.

In Unix, `^` and `$` are commonly used to match the beginning or end of a line, respectively. In this API, the regex metacharacters `^` and `$` ignore line terminators and only match at the beginning and the end, respectively, of the entire string. However, if you pass the `MULTILINE` flag into `Pattern.compile()`, these expressions match just after or just before, respectively, a line terminator; `$` also matches the very end of the string. Since the line ending is just an ordinary character, you can match it with `.` or similar expressions, and, if you want to know exactly where it is, `\n` or `\r` in the pattern match it as well. In other words, to this API, a newline character is just another character with no special significance. See the sidebar [Pattern.compile\(\) Flags](#). An example of newline matching is shown in [Example 4-6](#).

```

* Show line ending matching using regex class.
* @author Ian F. Darwin, ian@darwinsys.com
* @version $Id: ch04.xml,v 1.4 2004/05/04 20:11:27 ian Exp $
*/
public class NLMatch {
    public static void main(String[] argv) {

        String input = "I dream of engines\nmore engines, all day long";
        System.out.println("INPUT: " + input);
        System.out.println( );

        String[] patt = {
            "engines.more engines",
            "engines$"
        };

        for (int i = 0; i < patt.length; i++) {
            System.out.println("PATTERN " + patt[i]);

            boolean found;
            Pattern p1l = Pattern.compile(patt[i]);
            found = p1l.matcher(input).find( );
            System.out.println("DEFAULT match " + found);

            Pattern pml = Pattern.compile(patt[i],
                Pattern.DOTALL|Pattern.MULTILINE);
            found = pml.matcher(input).find( );
            System.out.println("MultiLine match " + found);
            System.out.println( );
        }
    }
}

```

If you run this code, the first pattern (with the wildcard character `.`) always matches, while the second pattern (with `$`) matches only when `MATCH_MULTILINE` is set.

```

> java NLMatch
INPUT: I dream of engines
more engines, all day long

PATTERN engines
more engines
DEFAULT match true
MULTILINE match: true

PATTERN engines$
DEFAULT match false
MULTILINE match: true

```

## Recipe 4.10. Program: Apache Logfile Parsing

The Apache web server is the world's leading web server and has been for most of the web's history. It is one of the world's best-known open source projects, and one of many fostered by the Apache Foundation. But the name Apache is a pun on the origins of the server; its developers began with the free NCSA server and kept hacking at it or "patching" it until it did what they wanted. When it was sufficiently different from the original, a new name was needed. Since it was now "a natchv server" the name Anache was chosen. One place this natchiness shows

through is in the log file format. Consider this entry:

```
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"
200 10450 "-" "Mozilla/4.6 [en] (X11; U; OpenBSD 2.8 i386; Nav)"
```

The file format was obviously designed for human inspection but not for easy parsing. The problem is that different delimiters are used: square brackets for the date, quotes for the request line, and spaces sprinkled all through. Consider trying to use a `StringTokenizer`; you might be able to get it working, but you'd spend a lot of time fiddling with it. However, this somewhat contorted regular expression<sup>[5]</sup> makes it easy to parse:

[5] You might think this would hold some kind of world record for complexity in regex competitions, but I'm sure it's been outdone many times.

```
^([\d.]+) (\S+) (\S+) \[([\w:/]+\s[+-]\d{4})\] \| "(.+?)" (\d{3}) (\d+) "([^\"]+)"
"([^\"]+)"
```

You may find it informative to refer back to [Table 4-2](#) and review the full syntax used here. Note in particular the use of the non-greedy quantifier `+` in `"(.+?)"` to match a quoted string; you can't just use `+` since that would match too much (up to the quote at the end of the line). Code to extract the various fields such as IP address, request, referer URL, and browser version is shown in [Example 4-7](#).

#### Example 4-7. LogRegExp.java

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

/**
 * Parse an Apache log file with Regular Expressions
 */
public class LogRegExp implements LogExample {

    public static void main(String argv[]) {

        String logEntryPattern =
            "^([\d.]+) (\S+) (\S+) \[([\w:/]+\s[+-]\d{4})\] \| "(.+?)" (\d{3})
            (\d+) \| "([^\"]+)" "([^\"]+)" ";
        System.out.println("Using regex Pattern:");
        System.out.println(logEntryPattern);

        System.out.println("Input line is:");
        System.out.println(logEntryLine);

        Pattern p = Pattern.compile(logEntryPattern);
        Matcher matcher = p.matcher(logEntryLine);
        if (!matcher.matches() ||
            NUM_FIELDS != matcher.groupCount() ) {
            System.err.println("Bad log entry (or problem with regex?):");
            System.err.println(logEntryLine);
            return;
        }
        System.out.println("IP Address: " + matcher.group(1));
        System.out.println("Date&Time: " + matcher.group(4));
        System.out.println("Request: " + matcher.group(5));
        System.out.println("Response: " + matcher.group(6));
        System.out.println("Bytes Sent: " + matcher.group(7));
        if (!matcher.group(8).equals("-"))
            System.out.println("Referer: " + matcher.group(8));
    }
}
```

```

        System.out.println("Browser: " + matcher.group(9));
    }
}

```

The implements clause is for an interface that just defines the input string; it was used in a demonstration to compare the regular expression mode with the use of a StringTokenizer. The source for both versions is in the online source for this chapter. Running the program against the sample input shown above gives this output:

```

Using regex Pattern:
^([\d.]+) (\S+) (\S+) \([([\w:/]+\s[+-]\d{4})\] "(.+)" (\d{3}) (\d+) "([^\"]+)"
"([^\"]+)"
Input line is:
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"
200 10450 "-" "Mozilla/4.6 [en] (X11; U; OpenBSD 2.8 i386; Nav)"
IP Address: 123.45.67.89
Date&Time: 27/Oct/2000:09:27:09 -0400
Request: GET /java/javaResources.html HTTP/1.0
Response: 200
Bytes Sent: 10450
Browser: Mozilla/4.6 [en] (X11; U; OpenBSD 2.8 i386; Nav)

```

The program successfully parsed the entire log file format with one call to `matcher.matches()`.

## Recipe 4.11. Program: Data Mining

Suppose that I, as a published author, want to track how my book is selling in comparison to others. This information can be obtained for free just by clicking on the page for my book on any of the major bookseller sites, reading the sales rank number off the screen, and typing the number into a file—but that's too tedious. As I wrote in the book that this example looks for, "computers get paid to extract relevant information from files; people should not have to do such mundane tasks." This program uses the Regular Expressions API and, in particular, newline matching to extract a value from an HTML page on the hypothetical QuickBookShops.web web site. It also reads from a URL object (see [Recipe 18.7](#)). The pattern to look for is something like this (bear in mind that the HTML may change at any time, so I want to keep the pattern fairly general):

```

<b>QuickBookShop.web Sales Rank: </b>
26,252
</font><br>

```

As the pattern may extend over more than one line, I read the entire web page from the URL into a single long string using my `FileIO.readerToString()` method (see [Recipe 10.8](#)) instead of the more traditional line-at-a-time paradigm. I then plot a graph using an external program (see [Recipe 26.1](#)); this could (and should) be changed to use a Java graphics program (see [Recipe 13.13](#) for some leads). The complete program is shown in [Example 4-8](#).

### Example 4-8. BookRank.java

```

// Standard imports not shown

```

```

import com.darwinsys.io.FileIO;
import com.darwinsys.util.FileProperties;

/** Graph of a book's sales rank on a given bookshop site.
 * @author Ian F. Darwin, http://www.darwinsys.com/, Java Cookbook
author,
 * originally translated fairly literally from Perl into Java.
 * @author Patrick Killelea <p@patrick.net>: original Perl version,
 * from the 2nd edition of his book "Web Performance Tuning".
 * @version $Id: ch04.xml,v 1.4 2004/05/04 20:11:27 ian Exp $
 */
public class BookRank {
    public final static String DATA_FILE = "book.sales";
    public final static String GRAPH_FILE = "book.png";

    /** Grab the sales rank off the web page and log it. */
    public static void main(String[] args) throws Exception {

        Properties p = new FileProperties(
            args.length == 0 ? "bookrank.properties" : args[1]);
        String title = p.getProperty("title", "NO TITLE IN PROPERTIES");
        // The url must have the "isbn=" at the very end, or otherwise
        // be amenable to being string-catted to, like the default.
        String url = p.getProperty("url", "http://test.ing/test.cgi?isbn=");
        // The 10-digit ISBN for the book.
        String isbn = p.getProperty("isbn", "0000000000");
        // The regex pattern (MUST have ONE capture group for the number)
        String pattern = p.getProperty("pattern", "Rank: (\\d+)");

        // Looking for something like this in the input:
        //      <b>QuickBookShop.web Sales Rank: </b>
        //      26,252
        //      </font><br>

        Pattern r = Pattern.compile(pattern);

        // Open the URL and get a Reader from it.
        BufferedReader is = new BufferedReader(new InputStreamReader(
            new URL(url + isbn).openStream( ));
        // Read the URL looking for the rank information, as
        // a single long string, so can match regex across multi-lines.
        String input = FileIO.readerToString(is);
        // System.out.println(input);

        // If found, append to sales data file.
        Matcher m = r.matcher(input);
        if (m.find( )) {
            PrintWriter pw = new PrintWriter(
                new FileWriter(DATA_FILE, true));
            String date = // 'date +%m %d %H %M %S %Y'`;
                new SimpleDateFormat("MM dd hh mm ss yyyy ").
                    format(new Date( ));
            // Paren 1 is the digits (and maybe ','s) that matched; remove comma
            Matcher noComma = Pattern.compile(",").matcher(m.group(1));
            pw.println(date + noComma.replaceAll(""));
            pw.close( );
        } else {
            System.err.println("WARNING: pattern `" + pattern +
                "' did not match in `" + url + isbn + "!!");
        }

        // Whether current data found or not, draw the graph, using
        // external plotting program against all historical data.
        // Could use gnuplot, R, any other math/graph program.
        // Better yet: use one of the Java plotting APIs.

        String gnuplot cmd =

```

```

        "set term png\n" +
        "set output \"" + GRAPH_FILE + "\"\n" +
        "set xdata time\n" +
        "set ylabel \"Book sales rank\"\n" +
        "set bmargin 3\n" +
        "set logscale y\n" +
        "set yrange [1:60000] reverse\n" +
        "set timefmt \"%m %d %H %M %S %Y\"\n" +
        "plot \"" + DATA_FILE +
            "\" using 1:7 title \"" + title + "\" with lines\n"
    ;

    Process proc = Runtime.getRuntime( ).exec("/usr/local/bin/gnuplot");
    PrintWriter gp = new PrintWriter(proc.getOutputStream( ));
    gp.print(gnuplot_cmd);
    gp.close( );
}
}

```

## Recipe 4.12. Program: Full Grep

Now that we've seen how the regular expressions package works, it's time to write Grep2, a full-blown version of the line-matching program with option parsing. [Table 4-3](#) lists some typical command-line options that a Unix implementation of grep might include.

Table 4-3. Grep command-line options	
Option	Meaning
-c	Count only: don't print lines, just count them
-C	Context; print some lines above and below each line that matches (not implemented in this version; left as an exercise for the reader)
-f pattern	Take pattern from file named after -f instead of from command line
-h	Suppress printing filename ahead of lines
-i	Ignore case
-l	List filenames only: don't print lines, just the names they're found in
-n	Print line numbers before matching lines
-s	Suppress printing certain error messages
-v	Invert: print only lines that do NOT match the pattern



```

        case 'f':
            try {
                BufferedReader b = new BufferedReader
                pattern = b.readLine( );
                b.close( );
            } catch (IOException e) {
                System.err.println("Can't read pattern file " +
                System.exit(1);
            }
            break;
        case 'h':
            args.set('H');
            break;
        case 'i':
            args.set('I');
            break;
        case 'l':
            args.set('L');
            break;
        case 'n':
            args.set('N');
            break;
        case 's':
            args.set('S');
            break;
        case 'v':
            args.set('V');
            break;
    }
}

int ix = go.getOptInd( );

if (pattern == null)
    pattern = argv[ix-1];

Grep2 pg = new Grep2(pattern, args);

if (argv.length == ix)
    pg.process(new InputStreamReader(System.in), "(standard input)");
else
    for (int i=ix; i<argv.length; i++) {
        try {
            pg.process(new FileReader(argv[i]), argv[i]);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

/** Construct a Grep2 object.
// compile the regular expression
if (args.get('C'))
    countOnly = true;
if (args.get('H'))
    dontPrintFileName = true;
if (args.get('I'))
    ignoreCase = true;
if (args.get('L'))
    listOnly = true;
if (args.get('N'))
    numbered = true;
if (args.get('S'))
    silent = true;
if (args.get('V'))
    inVert = true;
int caseMode = ignoreCase ? Pattern.UNICODE_CASE | Pattern.CASE_INSENSITIVE : 0;
pattern = Pattern.compile(patt, caseMode);
}

```

```

/** Do the work of scanning one file
 * @param ifile Reader Reader object already open
 * @param fileName String Name of the input file
 */
public void process(Reader ifile, String fileName) {

    String line;
    int matches = 0;

    try {
        d = new BufferedReader(ifile);

        while ((line = d.readLine( )) != null) {
            if (pattern.match(line)) {
                if (countOnly)
                    matches++;
                else {
                    if (!dontPrintFileName)
                        System.out.print(fileName + ": ");
                    System.out.println(line);
                }
            } else if (invert) {
                System.out.println(line);
            }
        }
        if (countOnly)
            System.out.println(matches + " matches in " + fileName);
        d.close( );
    } catch (IOException e) { System.err.println(e); }
}
}

```