8

Java

Java didn't come with a regex package until Java 1.4, so early programmers had to do without regular expressions. Over time, many programmers independently developed Java regex packages of varying degrees of quality, functionality, and complexity. With the early-2002 release of Java 1.4, Sun entered the fray with their `java.util.regex` package. In preparing this chapter, I looked at Sun's package, and a few others (detailed starting on page 372). So which one is best? As you'll soon see, there can be many ways to judge that.

In This Chapter Before looking at what's in this chapter, it's important to mention what's *not* in this chapter. In short, this chapter doesn't restate everything from Chapters 1 through 6. I understand that some readers interested only in Java may be inclined to start their reading with this chapter, and I want to encourage them not to miss the benefits of the preface and the earlier chapters: Chapters 1, 2, and 3 introduce basic concepts, features, and techniques involved with regular expressions, while Chapters 4, 5, and 6 offer important keys to regex understanding that directly apply to every Java regex package that I know of.

As for this chapter, it has several distinct parts. The first part, consisting of "Judging a Regex Package" and "Object Models," looks abstractly at some concepts that help you to understand an unfamiliar package more quickly, and to help judge its suitability for your needs. The second part, "Packages, Packages, Packages," moves away from the abstract to say a few words about the specific packages I looked at while researching this book. Finally, we get to the real fun, as the third part talks in specifics about two of the packages, Sun's `java.util.regex` and Jakarta's ORO package.

Judging a Regex Package

The first thing most people look at when judging a regex package is the regex flavor itself, but there are other technical issues as well. On top of that, "political" issues like source code availability and licensing can be important. The next sections give an overview of some points of comparison you might use when selecting a regex package.

Technical Issues

Some of the technical issues to consider are:

- **Engine Type?**   Is the underlying engine an NFA or DFA? If an NFA, is it a POSIX NFA or a Traditional NFA? (See Chapter 4 ☞ 143)

- **Rich Flavor?**   How full-featured is the flavor? How many of the items on page 113 are supported? Are they supported well? Some things are more important than others: lookaround and lazy quantifiers, for example, are more important than possessive quantifiers and atomic grouping, because lookaround and lazy quantifiers can't be mimicked with other constructs, whereas possessive quantifiers and atomic grouping can be mimicked with lookahead that allows capturing parentheses.

- **Unicode Support?**   How well is Unicode supported? Java strings support Unicode intrinsically, but does ⌜\w⌟ know which Unicode characters are "word" characters? What about ⌜\d⌟ and ⌜\s⌟? Does ⌜\b⌟ understand Unicode? (Does its idea of a word character match ⌜\w⌟'s idea of a word character?) Are Unicode properties supported? How about blocks? Scripts? (☞ 119) Which version of Unicode's mappings do they support: Version 3.0? Version 3.1? Version 3.2? Does case-insensitive matching work properly with the full breadth of Unicode characters? For example, does a case-insensitive 'ß' really match 'SS'? (Even in lookbehind?)

- **How Flexible?**   How flexible are the mechanics? Can the regex engine deal only with `String` objects, or the whole breadth of `CharSequence` objects? Is it easy to use in a multi-threaded environment?

- **How Convenient?**   The raw engine may be powerful, but are there extra "convenience functions" that make it easy to do the common things without a lot of cumbersome overhead? Does it, borrowing a quote from Perl, "make the easy things easy, and the hard things possible?"

- **JRE Requirements?**   What version of the JRE does it require? Does it need the latest version, which many may not be using yet, or can it run on even an old (and perhaps more common) JRE?

- **Efficient?** How efficient is it? The length of Chapter 6 tells you how much there is to be said on this subject. How many of the optimizations described there does it do? Is it efficient with memory, or does it bloat over time? Do you have any control over resource utilization? Does it employ lazy evaluation to avoiding computing results that are never actually used?

- **Does it Work?** When it comes down to it, does the package work? Are there a few major bugs that are "deal-breakers?" Are there many little bugs that would drive you crazy as you uncover them? Or is it a bulletproof, rock-solid package that you can rely on?

Of course, this list just the tip of the iceberg—each of these bullet points could be expanded out to a full chapter on its own. We'll touch on them when comparing packages later in this chapter.

Social and Political Issues

Some of the non-technical issues to consider are:

- **Documented?** Does it use Javadoc? Is the documentation complete? Correct? Approachable? Understandable?

- **Maintained?** Is the package still being maintained? What's the turnaround time for bugs to be fixed? Do the maintainers really care about the package? Is it being enhanced?

- **Support and Popularity?** Is there official support, or an active user community you can turn to for reliable support (and that you can provide support to, once you become skilled in its use)?

- **Ubiquity?** Can you assume that the package is available everywhere you go, or do you have to include it whenever you distribute your programs?

- **Licensing?** *May* you redistribute it when you distribute your programs? Are the terms of the license something you can live with? Is the source code available for inspection? *May* you redistribute modified versions of the source code? *Must* you?

Well, there are certainly a lot of questions. Although this book can give you the answers to some of them, it can't answer the most important question: *which is right for you?* I make some recommendations later in this chapter, but only you can decide which is best for you. So, to give you more background upon which to base your decision, let's look at one of the most basic aspects of a regex package: its object model.

Object Models

When looking at different regex packages in Java (or in any object-oriented language, for that matter), it's amazing to see how many different object models are used to achieve essentially the same result. An object model is the set of class structures through which regex functionality is provided, and can be as simple as one object of one class that's used for everything, or as complex as having separate classes and objects for each sub-step along the way. There is not an object model that stands out as the clear, obvious choice for every situation, so a lot of variety has evolved.

A Few Abstract Object Models

Stepping back a bit now to think about object models helps prepare you to more readily grasp an unfamiliar package's model. This section presents several representative object models to give you a feel for the possibilities without getting mired in the details of an actual implementation.

Starting with the most abstract view, here are some tasks that need to be done in using a regular expression:

> **Setup** . . .
>   ❶ Accept a string as a regex; compile to an internal form.
>   ❷ Associate the regex with the target text.
> **Actually apply the regex** . . .
>   ❸ Initiate a match attempt.
> **See the results** . . .
>   ❹ Learn whether the match is successful.
>   ❺ Gain access to further details of a successful attempt.
>   ❻ Query those details (what matched, where it matched, etc.).

These are the steps for just one match attempt; you might repeat them from ❸ to find the next match in the target string.

Now, let's look at a few potential object models from among the infinite variety that one might conjure up. In doing so, we'll look at how they deal with matching ⌜`\s+(\d+)`⌟ to the string '`May 16, 1998`' to find out that '` 16`' is matched overall, and '`16`' matched within the first set of parentheses (within "group one"). Remember, the goal here is to merely get a general feel for some of the issues at hand—we'll see specifics soon.

An "all-in-one" model

In this conceptual model, each regular expression becomes an object that you then use for everything. It's shown visually in Figure 8-1 below, and in pseudo-code here, as it processes all matches in a string:

```
DoEverythingObj myRegex = new DoEverythingObj("\\s+(\\d+)"); // ❶
     ⋮
while (myRegex.findMatch("May 16, 1998")) { // ❷, ❸, ❹
    String matched = myRegex.getMatchedText();  // ❺
    String num     = myRegex.group(1);          // ❻
     ⋮
}
```

As with most models in practice, the compilation of the regex is a separate step, so it can be done ahead of time (perhaps at program startup), and used later, at which point most of the steps are combined together, or are implicit. A twist on this might be to clone the object after a match, in case the results need to be saved for a while.
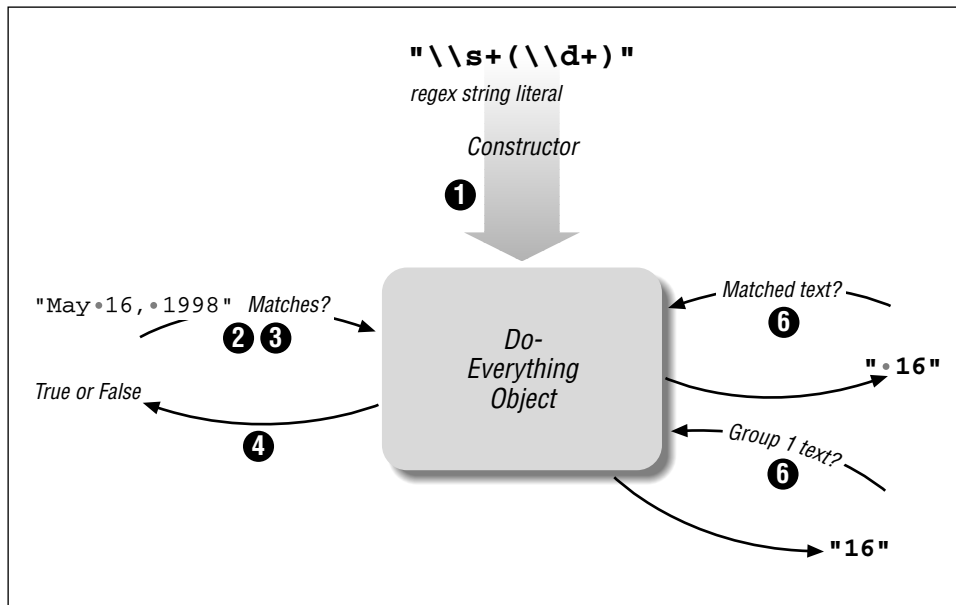

Figure 8-1: An "all-in-one" model

A "match state" model

This conceptual model uses two objects, a "Pattern" and a "Matcher." The Pattern object represents a compiled regular expression, while the Matcher object has all of the state associated with applying a Pattern object to a particular string. It's shown visually in Figure 8-2 below, and its use might be described as: "Convert a regex string to a Pattern object. Give a target string to the Pattern object to get a Matcher object that combines the two. Then, instruct the Matcher to find a match, and query the Matcher about the result." Here it is in pseudo-code:

```
PatternObj myPattern = new PatternObj("\\s+(\\d+)"); // ❶
    ⋮
MatcherObj myMatcher = myPattern.MakeMatcherObj("May 16, 1998"); // ❷
while (myMatcher.findMatch()) { // ❸ , ❹
    String matched = myMatcher.getMatchedText();  // ❺
    String num     = myMatcher.Group(1);          // ❻
      ⋮
}
```

This might be considered conceptually cleaner, since the compiled regex is in an immutable (unchangeable) object, and all state is in a separate object. However, it's not necessarily clear that the conceptual cleanliness translates to any practical benefit. One twist on this is to allow the Matcher to be reset with a new target string, to avoid having to make a new Matcher with each string checked.
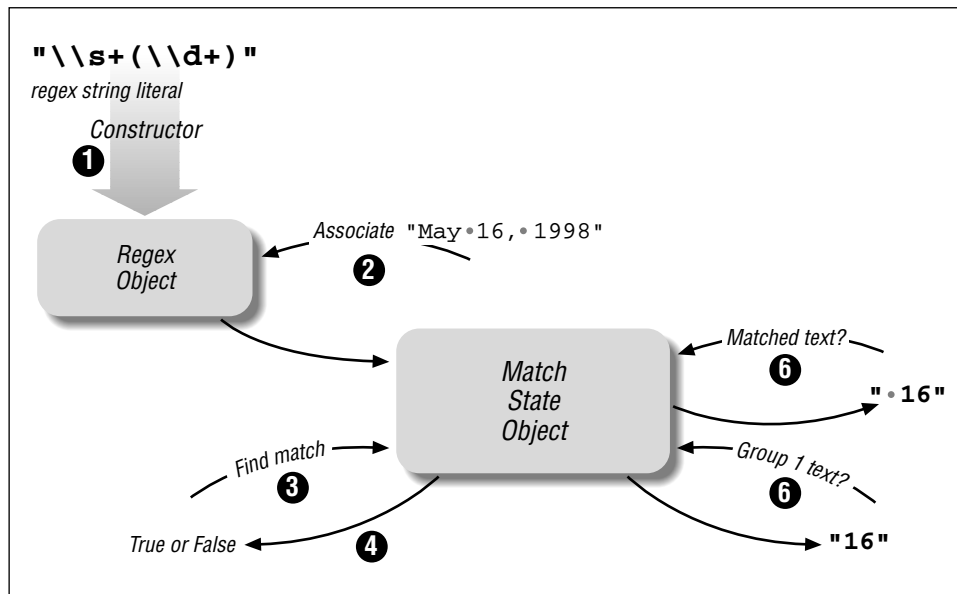

Figure 8-2: A "match state" model

A "match result" model

This conceptual model is similar to the "all-in-one" model, except that the result of a match attempt is not a Boolean, but rather a Result object, which you can then query for the specifics on the match. It's shown visually in Figure 8-3 below, and might be described as: "Convert a regex string to a Pattern object. Give it a target string and receive a Result object upon success. You can then query the Result object for specific." Here's one way it might be expressed it in pseudo-code:

```
PatternObj myPattern = new PatternObj("\\s+(\\d+)"); // ❶
       ⋮
ResultObj myResult = myPattern.findFirst("May 16, 1998"); // ❷, ❸, ❺
while (myResult.wasSuccessful()) { // ❹
    String matched = myResult.getMatchedText(); // ❻
    String num     = myResult.Group(1);         // ❻
       ⋮
    myResult = myPattern.findNext(); ❸, ❺
}
```

This compartmentalizes the results of a match, which might be convenient at times, but results in extra overhead when only a simple true/false result is desired. One twist on this is to have the Pattern object return `null` upon failure, to save the overhead of creating a Result object that just says "no match."
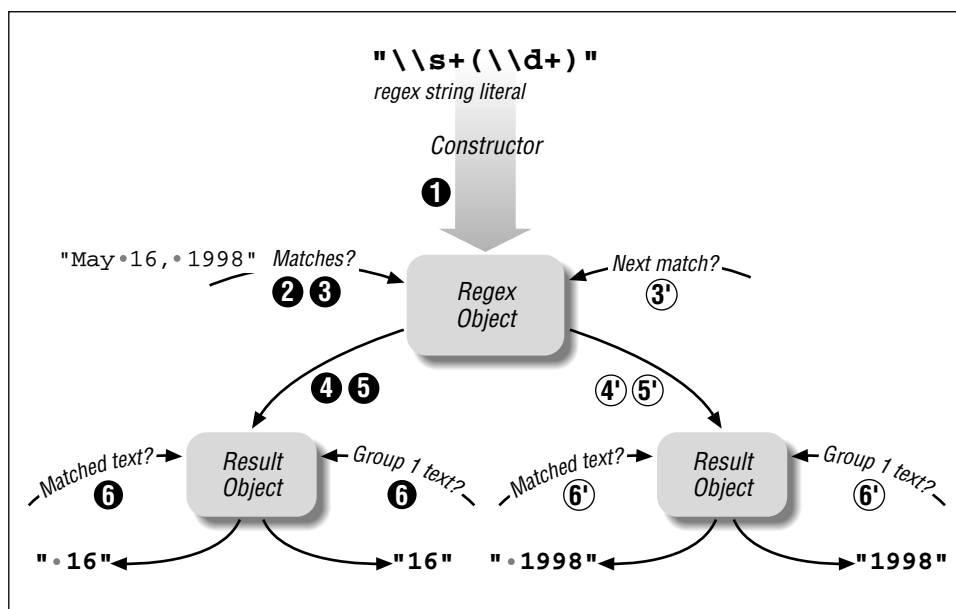

Figure 8-3: A "match result" model

Growing Complexity

These conceptual models are just the tip of the iceberg, but give you a feel for some of the differences you'll run into. They cover only simple matches — when you bring in search-and-replace, or perhaps string splitting (splitting a string into substrings separated by matches of a regex), it can become much more complex.

Thinking about search-and-replace, for example, the first thought may well be that it's a fairly simple task, and indeed, a simple "replace *this* with *that*" interface is easy to design. But what if the "that" needs to depend on what's matched by the "this," as we did many times in examples in Chapter 2 (☞ 67). Or what if you need to execute code upon every match, using the resulting text as the replacement? These, and other practical needs, quickly complicate things, which further increases the variety among the packages.

Packages, Packages, Packages

There are many regex packages for Java; the list that follows has a few words about those that I investigated while researching this book. (See this book's web page, `http://regex.info/`, for links). The table on the facing page gives a superficial overview of some of the differences among their flavors.

Sun

  `java.util.regex` Sun's own regex package, finally standard as of Java 1.4. It's a solid, actively maintained package that provides a rich Perl-like flavor. It has the best Unicode support of these packages. It provides all the basic functionality you might need, but has only minimal convenience functions. It matches against `CharSequence` objects, and so is extremely flexible in that respect. Its documentation is clear and complete. It is the all-around fastest of the engines listed here. This package is described in detail later in this chapter. Version Tested: 1.4.0.

  License: comes as part of Sun's JRE. Source code is available under SCSL (Sun Community Source Licensing)

IBM

  `com.ibm.regex` This is IBM's commercial regex package (although it's said to be similar to the `org.apache.xerces.utils.regex` package, which I did not investigate). It's actively maintained, and provides a rich Perl-like flavor, although is somewhat buggy in certain areas. It has very good Unicode support. It can match against `char[]`, `CharacterIterator`, and `String`. Overall, not quite as fast as Sun's package, but the only other package that's in the same class.
  Version Tested: 1.0.0.
  License: commercial product

Table 8-1: Superficial Overview of Some Java Package Flavor Differences

Feature	Sun	IBM	ORO	JRegex	Pat	GNU	Regexp
Basic Functionality							
Engine type	NFA	NFA	NFA	NFA	NFA	POSIX NFA	NFA
Deeply-nested parens	✓	✓	✓	✓	✓	✓	
dot doesn't match:	various	various	\n	\n , \r	\n	<u>\r\n</u>	\n
\s includes [\t\r\n\f]	✓	✓	✓			✓	✓
\w includes underscore	✓	✓	✓	✓	✓	✓	
Class set operators	✓	◿					
POSIX [[:⋯:]]	◿	◿	✓				
Metacharacter Support							
\A,\z,\Z	\A,\Z	\A,\z,\Z	\A,\z,\Z	\A,\z,\Z	\A,\Z	\A,\Z	
\G	✓	✓	✗		✓		
(?#⋯)		✓	✓	✓	✓	✓	
Octal escapes	✓		✓	✓	✓	✓	
2-, 4-, 6-digit hex escapes	2, 4	2, 4, 6	2	2, 4, 6	2		2, 4
Lazy quantifiers	✓	✓	✓	✓	✓	✓	✓
Atomic grouping		✓		✓			
Possessive quantifiers	✓						
Word boundaries	\b	\b	\b	\< \b \>	\b	\< \>	✗
Non-word boundaries	✓	✓	✓	✓	✗		✗
\Q⋯\E	✓					✗	
(*if then*|*else*) conditional		✓		✓			
Non-capturing parens	✓	✓	✓	✓	✓	✓	
Lookahead	✓	✓	✓	✓	✓	✓	
Lookbehind	✓	✗		✓	◿		
(?*mod*)	✓	✗	✓	✓	✓		
(?-*mod*:⋯)	✓	✗	✓	✓	✗		
(?*mod*:⋯)	✓	✗		✓	✓		
Unicode-Aware Metacharacters							
Unicode properties	✓	✓		✓			
Unicode blocks	✓	✓		✓			
dot, ^, $	✓	✓					
\w		✓	✓	✓		✓	✓
\d		✓	✓	✓		✓	✓
\s		✓	◿	✓		◿	◿
Word boundaries	✓	✓	✓	✓	✗	✓	✓
✓ - supported ◿ - partial support ✗ - supported, but buggy (Version info ☞372)							

ORO

> `org.apache.oro.text.regex` The Apache Jakarta project has two unrelated regex packages, one of which is "Jakarta-ORO." It actually contains multiple regex engines, each targeting a different application. I looked at one engine, the very popular `Perl5Compiler` matcher. It's actively maintained, and solid, although its version of a Perl-like flavor is much less rich than either the Sun or the IBM packages. It has minimal Unicode support. Overall, the regex engine is notably slower than most other packages. Its ⌜`\G`⌟ is broken. It can match against `char[]` and `String`.
>
> One of its strongest points is that it has a vast, modular structure that exposes almost all of the mechanics that surround the engine (the transmission, search-and-replace mechanics, etc.) so advanced users can tune it to suit their needs, but it also comes replete with a fantastic set of convenience functions that makes it one of the easiest packages to work with, particularly for those coming from a Perl background (or for those having read Chapter 2 of this book). This is discussed in more detail later in this chapter.
>
> Version Tested: 2.0.6.
>
> License: ASL (Apache Software License)

JRegex

> `jregex` Has the same object model as Sun's package, with a fairly rich Perl-like feature set. It has good Unicode support. Its speed places it is in the middle of the pack.
> Version Tested: v1.01
> License: GNU-like

Pat

> `com.stevesoft.pat` It has a fairly rich Perl-like flavor, but no Unicode support. Very haphazard interface. It has provisions for modifying the regex flavor on the fly. Its speed puts it on the high end of the middle of the pack.
> Version Tested: 1.5.3
> License: GNU LGPL (GNU Lesser General Public License)

GNU

> `gnu.regexp` The more advanced of the two "GNU regex packages" for Java. (The other, `gnu.rex`, is a very small package providing only the most bare-bones regex flavor and support, and is not covered in this book.) It has some Perl-like features, and minimal Unicode support. It's very slow. It's the only package with a POSIX NFA (although its POSIXness is a bit buggy at times).
> Version Tested: 1.1.4
> License: GNU LGPL (GNU Lesser General Public License)

Regexp

> `org.apache.regexp` This is the other regex package under the umbrella of the Apache Jakarta project. It's somewhat popular, but quite buggy. It has the fewest features of the packages listed here. Its overall speed is on par with ORO. Not actively maintained. Minimal Unicode support.
> Version Tested: 1.2
> License: ASL (Apache Software License)

Why So Many "Perl5" Flavors?

The list mentions "Perl-like" fairly often; the packages themselves advertise "Perl5 support." When version 5 of Perl was released in 1994 (☞ 89), it introduced a new level of regular-expression innovation that others, including Java regex developers, could well appreciate. Perl's regex flavor is powerful, and its adoption by a wide variety of packages and languages has made it somewhat of a de facto standard.

However, of the many packages, programs, and languages that claim to be "Perl5 compliant," none truly are. Even Perl itself differs from version to version as new features are added and bugs are fixed. Some of the innovations new with early 5.x versions of Perl were non-capturing parentheses, lazy quantifiers, lookahead, inline mode modifiers like ⌜`(?i)`⌟, and the `/x` free-spacing mode (all discussed in Chapter 3). Packages supporting only these features claim a "Perl5" flavor, but miss out on later innovations, such as lookbehind, atomic grouping, and conditionals.

There are also times when a package doesn't limit itself to only "Perl5" enhancements. Sun's package, for example, supports possessive quantifiers, and both Sun and IBM support character class set operations. Pat offers an innovative way to do lookbehind, and a way to allow matching of simple arbitrarily nested constructs.

Lies, Damn Lies, and Benchmarks

It's probably a common twist on Sam Clemens' famous "lies, damn lies, and statistics" quote, but when I saw its use with "benchmarks" in a paper from Sun while doing research for this chapter, I knew it was an appropriate introduction for this section. In researching these seven packages, I've run literally thousands of benchmarks, but the only fact that's clearly emerged is that there are no clear conclusions.

There are several things that cloud regex benchmarking with Java. First, there are language issues. Recall the benchmarking discussion from Chapter 6 (☞ 234), and the special issues that make benchmarking Java a slippery science at best (primarily, the effects of the Just-In-Time or Better-Late-Than-Never compiler). In doing these benchmarks, I've made sure to use a server VM that was "warmed up" for the benchmark (see "BLTN" ☞ 235), to show the truest results.

Then there are regex issues. Due to the complex interactions of the myriad of optimizations like those discussed in Chapter 6, a seemingly inconsequential change while trying to test one feature might tickle the optimization of an unrelated feature, anonymously skewing the results one way or the other. I did many (many!) very specific tests, usually approaching an issue from multiple directions, and so I believe I've been able to get meaningful results . . . but one never truly knows.

Warning: Benchmark results can cause drowsiness!

Just to show how slippery this all can be, recall that I judged the two Jakarta packages (ORO and Regexp) to be roughly comparable in speed. Indeed, they finished equally in some of the many benchmarks I ran, but for the most part, one generally ran at least twice the speed of the other (sometimes 10× or 20× the speed). But which was "one" and which "the other" changed depending upon the test.

For example, I targeted the speed of greedy and lazy quantifiers by applying ⌜`^.*:`⌝ and ⌜`^.*?:`⌝ to a very long string like '⋯`xxx:x`'. I expected the greedy one to be faster than the lazy one with this type of string, and indeed, it's that way for every package, program, and language I tested . . . except one. For whatever reason, Jakarta's Regexp's ⌜`^.*:`⌝ performed 70% slower than its ⌜`^.*?:`⌝. I then applied the same expressions to a similarly long string, but this time one like '`x:xxx`⋯' where the '`:`' is near the beginning. This should give the lazy quantifier an edge, and indeed, with Regexp, the expression with the lazy quantifier finished 670× faster than the greedy. To gain more insight, I applied ⌜`^[^:]*:`⌝ to each string. This should be in the same ballpark, I thought, as the lazy version, but highly contingent upon certain optimizations that may or may not be included in the engine. With Regexp, it finished the test a bit slower than the lazy version, for both strings.

Does the previous paragraph make your eyes glaze over a bit? Well, it discusses just six tests, and for only one regex package — we haven't even started to compare these Regexp results against ORO or any of the other packages. When compared against ORO, it turns out that Regexp is about 10× slower with four of the tests, but about 20× faster with the other two! It's faster with ⌜`^.*?:`⌝ and ⌜`^[^:]*:`⌝ applied to the long string with '`:`' at the front, so it seems that Regexp does poorly (or ORO does well) when the engine must walk through a lot of string, and that the speeds are reversed when the match is found quickly.

Are you eyes completely glazed over yet? Let's try the same set of six tests, but this time on short strings instead of very long ones. It turns out that Regexp is faster — three to ten times faster — than ORO for *all* of them. Okay, so what does this tell us? Perhaps that ORO has a lot of clunky overhead that overshadows the actual match time when the matches are found quickly. Or perhaps it means that Regexp is generally much faster, but has an inefficient mechanism for accessing the target string. Or perhaps it's something else altogether. I don't know.

Another test involved an "exponential match" (☞ 226) on a short string, which tests the basic churning of an engine as it tracks and backtracks. These tests took a long time, yet Regexp tended to finish in half the time of ORO. There just seems to be no rhyme nor reason to the results. Such is often the case when benchmarking something as complex as a regex engine.

And the winner is . . .

The mind-numbing statistics just discussed take into account only a small fraction of the many, varied tests I did. In looking at them all for Regexp and ORO, one package does not stand out as being faster overall. Rather, the good points and bad points seem to be distributed fairly evenly between the two, so I (perhaps somewhat arbitrarily) judge them to be about equal.

Adding the benchmarks from the five other packages into the mix results in a lot of drowsiness for your author, and no obviously clear winner, but overall, Sun's package seems to be the fastest, followed closely by IBM's. Following in a group somewhat behind are Pat, Jregex, Regexp, and ORO. The GNU package is clearly the slowest.

The overall difference between Sun and IBM is not so obviously clear that another equally comprehensive benchmark suite wouldn't show the opposite order if the suite happened to be tweaked slightly differently than mine. Or, for that matter, it's entirely possible that someone looking at all my benchmark data would reach a different conclusion. And, of course, the results could change drastically with the next release of any of the packages or virtual machines (and may well have, by the time you read this). It's a slippery science.

In general, Sun did most things very well, but it's missing a few key optimizations, and some constructs (such as character classes) are much slower than one would expect. Over time, these will likely be addressed by Sun (and in fact, the slowness of character classes is slated to be fixed in Java 1.4.2). The source code is available if you'd like to hack on it as well; I'm sure Sun would appreciate ideas and patches that improve it.

Recommendations

There are many reasons one might choose one package over another, but Sun's `java.util.regex` package—with its high quality, speed, good Unicode support, advanced features, and future ubiquity—is a good recommendation. It comes integrated as part of Java 1.4: `String.matches()`, for example, checks to see whether the string can be completely matched by a given regex.

`java.util.regex`'s strengths lie in its core engine, but it doesn't have a good set of "convenience functions," a layer that hides much of the drudgery of bit-shuffling behind the scenes. ORO, on the other hand, while its core engine isn't as strong, does have a strong support layer. It provides a very convenient set of functions for casual use, as well as the core interface for specialized needs. ORO is designed to allow multiple regex core engines to be plugged in, so the combination of `java.util.regex` with ORO sounds very appealing. I've talked to the ORO developer, and it seems likely that this will happen, so the rest of this chapter looks at Sun's `java.util.regex` and ORO's interface.

Sun's Regex Package

Sun's regex package, `java.util.regex`, comes standard with Java as of Version 1.4. It provides powerful and innovative functionality with an uncluttered (if somewhat simplistic) class interface to its "match state" object model discussed (☞ 370). It has fairly good Unicode support, clear documentation, and good efficiency.

We've seen examples of `java.util.regex` in earlier chapters (☞ 81, 95, 98, 217, 234). We'll see more later in this chapter when we look at its object model and how to actually put it to use, but first, we'll take a look at the regex flavor it supports, and the modifiers that influence that flavor.

Regex Flavor

`java.util.regex` is powered by a Traditional NFA, so the rich set of lessons from Chapters 4, 5, and 6 apply. Table 8-2 on the facing page summarizes its metacharacters. Certain aspects of the flavor are modified by a variety of match modes, turned on via flags to the various functions and factories, or turned on and off via ⌜`(?`*mods*–*mods*`)`⌟ and ⌜`(?`*mods*–*mods*`:`⋯`)`⌟ modifiers embedded within the regular expression itself. The modes are listed in Table 8-3 on page 380.

A regex flavor certainly can't be described with just a tidy little table, so here are some notes to augment Table 8-2:

- The table shows "raw" backslashes, not the doubled backslashes required when regular expressions are provided as Java string literals. For example, ⌜`\n`⌟ in the table must be written as `"\\n"` as a Java string. See "Strings as Regular Expressions" (☞ 101).

- With the `Pattern.COMMENTS` option (☞ 380), `#`⋯◫ sequences are taken as comments. (Don't forget to add newlines to multiline string literals, as in the sidebar on page 386.) Unescaped ASCII whitespace is ignored. **Note**: unlike most implementations that support this type of mode, comments and free whitespace *are* recognized within character classes.

Table 8-2: Overview of Sun's java.util.regex Flavor

Character Shorthands		
☞ 114	(c)	`\a \b \e \f \n \r \t \0`*octal* `\x`## `\u`#### `\c`*char*
Character Classes and Class-Like Constructs		
☞ 117	(c)	Classes: `[···]` `[^···]` (may contain class set operators ☞ 123)
☞ 118		Almost any character: *dot* (various meanings, changes with modes)
☞ 119	(c)	Class shorthands: `\w \d \s \W \D \S`
☞ 119	(c)	Unicode properties and blocks `\p{`*Prop*`}` `\P{`*Prop*`}`
Anchors and other Zero-Width Tests		
☞ 127		Start of line/string: `^` `\A`
☞ 127		End of line/string: `$` `\z` `\Z`
☞ 128		Start of current match: `\G`
☞ 131		Word boundary: `\b \B`
☞ 132		Lookaround: `(?=···)` `(?!···)` `(?<=···)` `(?<!···)`
Comments and Mode Modifiers		
☞ 133		Mode modifiers: `(?`*mods*-*mods*`)` Modifiers allowed: `x d s m i u`
☞ 134		Mode-modified spans: `(?`*mods*-*mods*`:···)`
☞ 112	(c)	Literal-text mode: `\Q···\E`
Grouping and Capturing		
☞ 135		Capturing parentheses: `(···)` `\1` `\2` . . .
☞ 136		Grouping-only parentheses: `(?:···)`
☞ 137		Atomic grouping: `(?>···)`
☞ 138		Alternation: `|`
☞ 139		Greedy quantifiers: `*` `+` `?` `{n}` `{n,}` `{x,y}`
☞ 140		Lazy quantifiers: `*?` `+?` `??` `{n}?` `{n,}?` `{x,y}?`
☞ 140		Possessive quantifiers: `*+` `++` `?+` `{n}+` `{n,}+` `{x,y}+`

(c) – may be used within a character class   (See text for notes on many items)

- `\b` is valid as a backspace only within a character class (outside, it matches a word boundary).

- `\x`## allows exactly two hexadecimal digits, e.g., ⌜`\xFC`ber⌝ matches 'über'.

- `\u`#### allows exactly four hexadecimal digits, e.g., ⌜`\u00FC`ber⌝ matches 'über', and ⌜\u20AC⌝ matches '€'.

- `\0`*octal* requires the leading zero, with one to three following octal digits.

- `\c`*char* is *case sensitive*, blindly *xor*ing the ordinal value of the following character with 64. This bizarre behavior means that, unlike any other regex flavor I've ever seen, \cA and \ca are different. Use uppercase letters to get the traditional meaning of \x01. As it happens, \ca is the same as \x21, matching '!'. (The case sensitivity is scheduled to be fixed in Java 1.4.2.)

Table 8-3: The java.util.regex Match and Regex Modes

Compile-Time Option	(?*mode*)	Description
`Pattern.UNIX_LINES`	d	Changes how *dot* and ⌈^⌋ match (☞ 382)
`Pattern.DOTALL`	s	Causes *dot* to match any character (☞ 110)
`Pattern.MULTILINE`	m	Expands where ⌈^⌋ and ⌈$⌋ can match (☞ 382)
`Pattern.COMMENTS`	x	Free-spacing and comment mode (☞ 72) (Applies even inside character classes)
`Pattern.CASE_INSENSITIVE`	i	Case-insensitive matching for ASCII characters
`Pattern.UNICODE_CASE`	u	Case-insensitive matching for non-ASCII characters
`Pattern.CANON_EQ`		Unicode "canonical equivalence" match mode (different encodings of the same character match as identical ☞ 107)

- **\w**, **\d**, and **\s** (and their uppercase counterparts) match only ASCII characters, and don't include the other alphanumerics, digits, or whitespace in Unicode. That is, \d is exactly the same as `[0-9]`, \w is the same as `[0-9a-zA-Z_]`, and \s is the same as `[•\t\n\f\r\x0B]` (\x0B is the little-used ASCII VT character).

  For full Unicode coverage, you can use Unicode properties (☞ 119): use \p{L} for \w, use \p{Nd} for \d, and use \p{Z} for \s. (Use the \P{⋯} version of each for \W, \D, and \S.)

- **\p{⋯}** and **\P{⋯}** support most standard Unicode properties and blocks. Unicode scripts are not supported. Only the short property names like \p{Lu} are supported—long names like \p{Lowercase_Letter} are not supported. (See the tables on pages 120 and 121.) One-letter property names may omit the braces: \pL is the same as \p{L}. Note, however, that the special composite property \p{L&} is not supported. Also, for some reason, \p{P} does not match characters matched by \p{Pi} and \p{Pf}. \p{C} doesn't match characters matched by \p{Cn}.

  \p{all} is supported, and is equivalent to (?s:.). \p{assigned} and \p{unassigned} are *not* supported: use \P{Cn} and \p{Cn} instead.

- This package understands Unicode blocks as of Unicode Version 3.1. Blocks added to or modified in Unicode since Version 3.1 are not known (☞ 108).

  Block names require the 'In' prefix (see the table on page 123), and only the raw form unadorned with spaces and underscores may be used. For example, \p{In_Greek_Extended} and \p{In Greek Extended} are not allowed; \p{InGreekExtended} is required.

- **$** and **\z** actually match line terminators when they should only match *at* the line terminators (for example, a pattern of `"(.*$)"` actually captures the line terminator). This is scheduled to be fixed in Java 1.4.1.

- **\G** matches the location where the current match started, despite the documentation's claim that it matches at the ending location of the previous match (☞ 128). ⌈\G⌋ is scheduled to be fixed (to agree with the documentation and match at the end of the previous match) in Java 1.4.1.

- The **\b** and **\B** word boundary metacharacters' idea of a "word character" is not the same as \w and \W's. The word boundaries understand the properties of Unicode characters, while \w and \W match only ASCII characters.

- Look*ahead* constructs can employ arbitrary regular expressions, but look*behind* is restricted to subexpressions whose possible matches are finite in length. This means, for example, that ⌈?⌋ is allowed within lookbehind, but ⌈*⌋ and ⌈+⌋ are not. See the description in Chapter 3, starting on page 132.

- At least until Java 1.4.2 is released, character classes with many elements are not optimized, and so are very slow; use ranges when possible (e.g., use `[0-9A-F]` instead of `[0123456789ABCDEF]`), and if there are characters or ranges that are likely to match more often than others, put them earlier in the class's list.

Using java.util.regex

The mechanics of wielding regular expressions with `java.util.regex` are fairly simple. Its object model is the "match state" model discussed on page 370. The functionality is provided with just three classes:

```
java.util.regex.Pattern
java.util.regex.Matcher
java.util.regex.PatternSyntaxException
```

Informally, I'll refer to the first two simply as "`Pattern`" and "`Matcher`". In short, the `Pattern` object is a compiled regular expression that can be applied to any number of strings, and a `Matcher` object is an individual instance of that regex being applied to a specific target string. The third class is the exception thrown upon the attempted compilation of an ill-formed regular expression.

Sun's documentation is sufficiently complete and clear that I refer you to it for the complete list of all methods for these objects (if you don't have the documentation locally, see `http://regex.info` for links). The rest of this section highlights just the main points.

Sun's java.util.regex "Line Terminators"

Traditionally, pre-Unicode regex flavors treat a newline specially with respect to *dot*, ⌜^⌟, ⌜$⌟, and ⌜\z⌟. However, the Unicode standard suggests the larger set of "line terminators" discussed in Chapter 3 (☞ 108). Sun's package supports a subset of these consisting of these five characters and one character sequence:

Character Codes	Nicknames		Description
U+000A	LF	\n	ASCII Line Feed
U+000D	CR	\r	ASCII Carriage Return
U+000D U+000A	CR/LF	\r\n	ASCII Carriage Return / Line Feed
U+0085	NEL		Unicode NEXT LINE
U+2028	LS		Unicode LINE SEPARATOR
U+2029	PS		Unicode PARAGRAPH SEPARATOR

This list is related to the *dot*, ⌜^⌟, ⌜$⌟, and ⌜\z⌟ metacharacters, but the relationships are neither constant (they change with modes), nor consistent (one would expect ⌜^⌟ and ⌜$⌟ to be treated similarly, but they are not).

Both the `Pattern.UNIX_LINES` and `Pattern.DOTALL` match modes (available also via ⌜(?d)⌟ and ⌜(?s)⌟) influence what *dot* matches.

⌜^⌟ can always match at the beginning of the string, but can match elsewhere under the `(?m)` `Pattern.MULTILINE` mode. It also depends upon the ⌜(?d)⌟ `Pattern.UNIX_LINES` mode.

⌜$⌟ and ⌜\z⌟ can always match at the end of the string, but they can also match just before certain string-ending line terminators. With the `Pattern.MULTILINE` mode, ⌜$⌟ can match after certain embedded line terminators as well. With Java 1.4.0, `Pattern.UNIX_LINES` does not influence ⌜$⌟ and ⌜\z⌟ in the same way (but it's slated to be fixed in 1.4.1 such that it does). The following table summarizes the relationships as of 1.4.0.

	LF	CR	CR/LF	NEL	LS	PS
Default action, without modifiers						
dot matches all but:	✓	ℐ		ℐ	ℐ	ℐ
^ matches at beginning of string only						
$ and **\z** match before string-ending:	✓	✓	✓		✓	✓
With `Pattern.MULTILINE` or (?m)						
^ matches after any:	✓	ℐ	ℐ	ℐ	ℐ	ℐ
$ matches before any:	✓	✓	✓		✓	✓
With `Pattern.DOTALL` or (?s)						
dot matches any character						

ℐ — does not apply if `Pattern.UNIX_LINES` or (?d) is in effect

Finally, note that there is a bug in Java 1.4.0 that is slated to be fixed in 1.4.1: ⌜$⌟ and ⌜\z⌟ actually *match* the line terminators, when present, rather than merely matching *at* line terminators.

Here's a complete example showing a simple match:

```
public class SimpleRegexTest {
  public static void main(String[] args)
  {
    String sampleText = "this is the 1st test string";
    String sampleRegex = "\\d+\\w+";
    java.util.regex.Pattern p = java.util.regex.Pattern.compile(sampleRegex);
    java.util.regex.Matcher m = p.matcher(sampleText);

    if (m.find()) {
      String matchedText = m.group();
      int    matchedFrom = m.start();
      int    matchedTo   = m.end();
      System.out.println("matched [" + matchedText + "] from " +
                          matchedFrom + " to " + matchedTo + ".");
    } else {
      System.out.println("didn't match");
    }
  }
}
```

This prints '**matched [1st] from 12 to 15.**'. As with all examples in this chapter, names I've chosen are in italic. Notice the Matcher object, after having been created by associating a Pattern object and a target string, is used to instigate the actual match (with its m.find() method), and to query the results (with m.group(), etc.).

The parts shown in bold can be omitted if

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

or perhaps

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

are inserted at the head of the program, just as with the examples in Chapter 3 (☞ 95). Doing so makes the code more manageable, and is the standard approach. The rest of this chapter assumes the import statement is always supplied. A more involved example is shown in the sidebar on page 386.

The Pattern.compile() Factory

A Pattern regular-expression object is created with Pattern.compile(⋯). The first argument is a string to be interpreted as a regular expression (☞ 101). Optionally, compile-time options shown in Table 8-3 on page 380 can be provided as a second argument. Here's a snippet that creates a Pattern object from the string in the variable sampleRegex, to be matched in a case-insensitive manner:

```
Pattern pat = Pattern.compile(sampleRegex,
                    Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
```

A call to Pattern.compile(⋯) can throw two kinds of exceptions: an invalid regular expression throws PatternSyntaxException, and an invalid option value throws IllegalArgumentException.

Pattern's `matcher(···)` method

A `Pattern` object offers some convenience methods we'll look at shortly, but for the most part, all the work is done through just one method: `matcher(···)`. It accepts a single argument: the string to search.[†] It doesn't actually apply the regex, but prepares the general `Pattern` object to be applied to a specific string. The `matcher(···)` method returns a `Matcher` object.

The `Matcher` Object

Once you've associated a regular expression with a target string by creating a `Matcher` object, you can instruct it to apply the regex in various ways, and query the results of that application. For example, given a `Matcher` object m, the call `m.find()` actually applies m's regex to its string, returning a Boolean indicating whether a match is found. If a match is found, the call `m.group()` returns a string representing the text actually matched.

The next sections list the various `Matcher` methods that actually apply a regex, followed by those that query the results.

Applying the regex

Here are the main `Matcher` methods for actually applying its regex to its string:

find()

> Applies the object's regex to the object's string, returning a Boolean indicating whether a match is found. If called multiple times, the next match is returned each time.

find(_offset_**)**

> If `find(···)` is given an integer argument, the match attempt starts from the given _offset_ number of characters from the start of the string. It throws `IndexOutOfBoundsException` if the _offset_ is negative or beyond the end of the string.

matches()

> This method returns a Boolean indicating whether the object's regex _exactly_ matches the object's string. That is, the regex is wrapped with an implied ⌜\A···\z⌟.[‡] This is also available via `String`'s `matches()` method. For example, `"123".matches("\\d+")` is true.

---

† Actually, `matcher`'s argument can be any object implementing the `CharSequence` interface (of which `String`, `StringBuffer`, and `CharBuffer` are examples). This provides the flexibility to apply regular expressions to a wide variety of data, including text that's not even kept in contiguous strings.

‡ Due to the bug with ⌜\z⌟ mentioned at the bottom of page 382, with version 1.4.0, the regex actually appears to be wrapped with an implied ⌜\A·**\z**⌟ instead.

lookingAt()

> Returns a Boolean indicating whether the object's regex matches the object's string *from its beginning*. That is, the regex applied with an implied ⌜\A⌟ at its start.

Querying the results

The following `Matcher` methods return information about a successful match. They throw `IllegalStateException` if the object's regex hasn't yet been applied to the object's string, or if the previous application was not successful. The methods that accept a *num* argument (referring to a set of capturing parentheses) throw `IndexOutOfBoundsException` when an invalid *num* is given.

group()

> Returns the text matched by the previous regex application.

groupCount()

> Returns the number of sets of capturing parentheses in the object's regex. Numbers up to this value can be used in the `group(`*num*`)` method, described next.

group(_num_**)**

> Returns the text matched by the *num*th set of capturing parentheses, or `null` if that set didn't participate in the match. A *num* of zero indicates the entire match, so `group(0)` is the same as `group()`.

start(_num_**)**

> Returns the offset, in characters, from the start of the string to the start of where the *num*th set of capturing parentheses matched. Returns `-1` if the set didn't participate in the match.

start()

> The offset to the start of the match; this is the same as `start(0)`.

end(_num_**)**

> Returns the offset, in characters, from the start of the string to the end of where the *num*th set of capturing parentheses matched. Returns `-1` if the set didn't participate in the match.

end()

> The offset to the end of the match; this is the same as `end(0)`.

Reusing `Matcher` objects for efficiency

The whole point of having separate compile and apply steps is to increase efficiency, alleviating the need to recompile a regex with each use (☞ 241). Additional efficiency can be gained by reusing `Matcher` objects when applying the same regex to new text. This is done with the `reset` method, described next.

CSV Parsing with java.util.regex

Here's the `java.util.regex` version of the CSV example from Chapter 6 (☞ 271). The regex has been updated to use possessive quantifiers (☞ 140) for a bit of extra efficiency.

First, we set up `Matcher` objects that we'll use in the actual processing. The '\n' at the end of each line is needed because we use ⌈#⋯⌋ comments, which end at a newline.

```
//Prepare the regexes we'll use

Pattern pCSVmain = Pattern.compile(
    "  \\G(?:^|,)                                 \n"+
    "  (?:                                        \n"+
    "     # Either a double-quoted field...       \n"+
    "     \" # field's opening quote              \n"+
    "       (  (?> [^\"]*+ ) (?> \"\" [^\"]*+ )*+  )   \n"+
    "     \" # field's closing quote              \n"+
    "   # ... or ...                              \n"+
    "   |                                         \n"+
    "     # ... some non-quote/non-comma text ... \n"+
    "     ( [^\",]*+ )                            \n"+
    "  )                                          \n",
    Pattern.COMMENTS);
Pattern pCSVquote = Pattern.compile("\"\"");
// Now create Matcher objects, with dummy text, that we'll use later.
Matcher mCSVmain  = pCSVmain.matcher("");
Matcher mCSVquote = pCSVquote.matcher("");
```

Then, to parse the string in `csvText` as CSV text, we use those `Matcher` objects to actually apply the regex and use the results:

```
mCSVmain.reset(csvText); // Tie the target text to the mCSVmain object
while ( mCSVmain.find() )
{
    String field; // We'll fill this in with $1 or $2 . . .
    String second = mCSVmain.group(2);
    if ( second != null )
        field = second;
    else {
        // If $1, must replace paired double-quotes with one double quote
        mCSVquote.reset(mCSVmain.group(1));
        field = mCSVquote.replaceAll("\"");
    }
    // We can now work with field . . .
    System.out.println("Field [" + field + "]");
}
```

This is more efficient than the similar version shown on page 217 for two reasons: the regex is more efficient (as per the Chapter 6 discussion), and that one `Matcher` object is reused, rather than creating and disposing of new ones each time (as per the discussion on page 385).

reset(_text_**)**

>This method reinitializes the `Matcher` object with the given `String` (or any object that implements a `CharSequence`), such that the next regex operation will start at the beginning of this text. This is more efficient than creating a new `Matcher` object (☞ 385). You can omit the argument to keep the current text, but to reset the match state to the beginning.

Reusing the `Matcher` object saves the Java mechanics of disposing of the old object and creating a new one, and requires only about one fourth the overhead of creating a new `Matcher` object.

In practice, you usually need only one `Matcher` object per regex, at least if you intend to apply the regex to only one string at a time, as is commonly the case. The sidebar on the facing page shows this in action. Dummy strings are immediately associated with each `Pattern` object to create the `Matcher` objects. It's okay to start with a dummy string because the object's `reset(⋯)` method is called with the real text to match against before the object is used further.

In fact, there's really no need to actually save the `Pattern` objects to variables, since they're not used except to create the `Matcher` objects. The lines:

```
Pattern pCSVquote = Pattern.compile("\"\"");
Matcher mCSVquote = mCSVquote.matcher("");
```

can be replaced by

```
Matcher mCSVquote = Pattern.compile("\"\"").matcher("");
```

thus eliminating the `pCSVquote` variable altogether.

Simple search and replace

You can implement search-and-replace operations using just the methods mentioned so far, but the `Matcher` object offers convenient methods to do simple search-and-replace for you:

replaceAll(_replacement_**)**

>The `Matcher` object is reset, and its regex is repeatedly applied to its string. The return value is a copy of the object's string, with any matches replaced by the *replacement* string.

>This is also available via a `String`'s `replaceAll` method:

```
string.replaceAll(regex, replacement);
```

>is equivalent to:

```
Pattern.compile(regex).matcher(string).replaceAll(replacement)
```

replaceFirst(_replacement_**)**

>The `Matcher` object is reset, and its regex is applied once to its string. The return value is a copy of the object's string, with the first match (if any) replaced by the *replacement* string.

This is also available via a `String`'s `replaceFirst` method, as just described with `replaceAll`.

With any of these functions, the *replacement* string receives special parsing:

- Instances of '`$1`', '`$2`', etc., within the replacement string are replaced by the text matched by the associated set of capturing parentheses. (`$0` is replaced by the entire text matched.)

  `IllegalArgumentException` is thrown if the character following the '`$`' is not an ASCII digit.

  Only as many digits after the '`$`' as "make sense" are used. For example, if there are three capturing parentheses, '`$25`' in the replacement string is interpreted as `$2` followed by the character '`5`'. However, in the same situation, '`$6`' in the replacement string throws `IndexOutOfBoundsException`.

- A backslash escapes the character that follows, so use '`···\$···`' in the replacement string to include a dollar sign in it. By the same token, use '`···\\···`' to get a backslash into the replacement value. (And if you're providing the replacement string as a Java string literal, that means you need `"···\\\\···"` to get a backslash into the replacement value.) Also, if there are, say, 12 sets of capturing parentheses and you'd like to include the text matched by the first set, followed by '2', you can use a replacement value of '`···$1\2···`'.

Advanced search and replace

Two additional methods provide raw access to `Matcher`'s search-and-replace mechanics. Together, they build a result in a `StringBuffer` that you provide. The first is called after each match, to fill the result with the replacement string, as well as the text between the matches. The second is called after all matches have been found, to tack on the text remaining after the final match.

`appendReplacement(`*stringBuffer***`,` ***replacement***`)`**

Called immediately after a regex has been successfully applied (e.g., with `find`), this method appends two strings to the given *stringBuffer*: first, it copies in the text of the original target string prior to the match. Then it appends the *replacement* string, as per the special processing described in the previous section.

For example, let's say we've got a `Matcher` object `m` that associates the regex ⌜`\w+`⌟ with the string '`-->one+test<--`'. The first time through this `while` loop:

```
while (m.find())
    m.appendReplacement(sb, "XXX")
```

the `find` matches the underlined portion of '`-->one+test<--`'. The call to `appendReplacement` fills the *stringBuffer* sb with the text before the match,

'-->', then bypasses what matched, instead appending the replacement string, 'XXX', to sb.

The second time through the loop, `find` matches '-->one+test<--'. The call to `appendReplacement` appends the text before the match, '+', then again appends the replacement string, 'XXX'.

This leaves sb with '-->XXX+XXX', and the original target string within the m object marked at '-->one+test<--'.

appendTail(*stringBuffer***)**

Called after all matches have been found (or, at least, after the desired matches have been found — you can stop early if you like), this method appends the remaining text. Continuing the previous example,

```
m.appendTail(sb)
```

appends '<--' to sb. This leaves it with '-->XXX+XXX<--', completing the search and replace.

Here's an example showing how you might implement your own version of `replaceAll` using these. (Not that you'd want to, but it's illustrative.)

```
public static String replaceAll(Matcher m, String replacement)
{
    m.reset(); // Be sure to start with a fresh Matcher object
    StringBuffer result = new StringBuffer(); // We'll build the updated copy here
    while (m.find())
        m.appendReplacement(result, replacement);
    m.appendTail(result);
    return result.toString(); //Convert to a String and return
}
```

Here's a slightly more involved snippet, which prints a version of the string in the variable `metric`, with Celsius temperatures converted to Fahrenheit:

```
// Build a matcher to find numbers followed by "C" within the variable "Metric"
Matcher m = Pattern.compile("(\\d+(?:\\.(\\d+))?)C\\b").matcher(metric);

StringBuffer result = new StringBuffer(); // We'll build the updated copy here
while (m.find()) {
    float celsius = Float.parseFloat(m.group(1));   //Get the number, as a number
    int fahrenheit = (int) (celsius * 9/5 + 32);    //Convert to a Fahrenheit value
    m.appendReplacement(result, fahrenheit + "F"); //Insert it
}
m.appendTail(result);
System.out.println(result.toString()); //Display the result
```

For example, if the variable `metric` contains '**from 36.3C to 40.1C.**', it displays '**from 97F to 104F.**'.

Other `Pattern` *Methods*

In addition to the main `compile(⋯)` factories, the `Pattern` class contains some helper functions and methods that don't add new functionality, but make the current functionality more easily accessible.

`Pattern.matches(`*pattern****,*** *text***`)`**

> This static function returns a Boolean indicating whether the string *pattern* can match the `CharSequence` (e.g., `String`) *text.* Essentially, this is:
>
> ```
> Pattern.compile(pattern).matcher(text).matches();
> ```
>
> If you need to pass compile options, or need to gain access to more information about the match than whether it was successful, you'll have to use the methods described earlier.

Pattern's split method, with one argument

`split(`*text***`)`**

> This `Pattern` method accepts *text* (a `CharSequence`) and returns an array of strings from *text* that are delimited by matches of the object's regex. This is also available via a `String`'s `split` method.

This trivial example

```
String[] result = Pattern.compile("\\.").split("209.204.146.22");
```

returns the array of four strings ('`209`', '`204`', '`146`', and '`22`') that are separated by the three matches of ⌈`\.`⌋ in the text. This simple example splits on only a single literal character, but you can split on an arbitrary regular expression. For example, you might approximate splitting a string into "words" by splitting on non-alphanumerics:

```
String[] result = Pattern.compile("\\W+").split(Text);
```

When given a string like '`What's␣up,␣␣Doc`' it returns the four strings ('`What`', '`s`', '`up`', and '`Doc`') delimited by the three matches of the regex. (If you had non-ASCII text, you'd probably want to use ⌈`\p{L}+`⌋, or perhaps ⌈`[^\p{L}\p{N}_]`⌋, as the regex, instead of ⌈`\W+`⌋ ☞ 380.)

Empty elements with adjacent matches

If the object's regex can match at the beginning of the *text*, the first string returned by `split` is an empty string (a valid string, but one that contains no characters). Similarly, if the regex can match two or more times in a row, empty strings are returned for the zero-length text "separated" by the adjacent matches. For example,

```
String[] result = Pattern.compile("\\s*,\\s*").split(", one, two , ,, 3");
```

splits on a comma and any surrounding whitespace, returning an array of five strings: an empty string, '`one`', '`two`', two empty strings, and '`3`'.

Finally, any empty strings that might appear at the *end* of the list are suppressed:

```
String[] result = Pattern.compile(":").split(":xx:");
```

This produces just two strings: an empty string and 'xx'. To keep trailing empty elements, use the two-argument version of split(···), described next.

Pattern's split method, with two arguments

split(*text,* *limit***)**

> This version of split(···) provides some control over how many times the Pattern's regex is applied, and what is done with trailing empty elements.

The *limit* argument takes on different meanings depending on whether it's less than zero, zero, or greater than zero.

Split with a limit less than zero
Any *limit* less than zero means to keep trailing empty elements in the array. Thus,

```
String[] result = Pattern.compile(":").split(":xx:", -1);
```

returns an array of three strings (an empty string, 'xx', and another empty string).

Split with a limit of zero
An explicit limit of zero is the same as if there were no limit given, i.e., trailing empty elements are suppressed.

Split with a limit greater than zero
With a *limit* greater than zero, split(···) returns an array of at most *limit* elements. This means that the regex is applied at most *limit*-1 times. (A limit of three, for example, requests *three* strings separated by *two* matches.)

After having matched *limit*-1 times, no further matches are checked, and the entire remainder of the string after the final match is returned as the last string in the array. For example, if you had a string with

```
Friedl,Jeffrey,Eric Francis,America,Ohio,Rootstown
```

and wanted to isolate just the three name components, you'd split the string into four parts (the three name components, and one final "everything else" string):

```
String[] NameInfo = Pattern.compile(",").split(Text, 4);
// NameInfo[0] is the family name
// NameInfo[1] is the given name
// NameInfo[2] is the middle name (or in my case, middle names)
// NameInfo[3] is everything else, which we don't need, so we'll just ignore it.
```

The reason to limit split in this way is for enhanced efficiency — why bother going through the work of finding the rest of the matches, creating new strings, making a larger array, etc., when there's no intention to use the results of that work? Supplying a limit allows just the required work to be done.

A Quick Look at Jakarta-ORO

Jakarta-ORO (from now on, just "ORO") is a vast, modular framework of mostly regex-related text-processing features containing a dizzying eight interfaces and 35+ classes. When first faced with the documentation, you can be intimidated until you realize that you can get an amazing amount of use out of it by knowing just one class, `Perl5Util`, described next.

ORO's `Perl5Util`

This ORO version of the example from page 383 shows how simple `Perl5Util` is to work with:

```
import org.apache.oro.text.perl.Perl5Util;

public class SimpleRegexTest {
  public static void main(String[] args)
  {
     String sampleText = "this is the 1st test string";
     Perl5Util engine = new Perl5Util();

     if (engine.match("/\\d+\\w+/", sampleText)) {
         String matchedText = engine.group(0);
         int    matchedFrom = engine.beginOffset(0);
         int    matchedTo   = engine.endOffset(0);
         System.out.println("matched [" + matchedText + "] from " +
                           matchedFrom + " to " + matchedTo + ".");
     } else {
         System.out.println("didn't match");
     }
  }
}
```

One class hides all the messy details about working with regular expressions behind a simple façade that somewhat mimics regular-expression use in Perl. Where Perl has

```
$input =~ /^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i
```

(from an example in Chapter 2 ☞ 48), ORO allows:

```
engine.match("/^([-+]?[0-9]+(\\.[0-9]*)?)\\s*([CF])$/i", input)
```

Where Perl then has

```
$InputNum = $1;   # Save to named variables to make the ...
$Type     = $3;   # ... rest of the program easier to read.
```

ORO provides for:

```
inputNum = engine.group(1);  // Save to named variables to make the ...
type     = engine.group(3);  // ... rest of the program easier to read.
```

If you're not familiar with Perl, the /⋯/i trappings may seem a bit odd, and they can be cumbersome at times, but it lowers the barrier to regex use about as low as

it can get in Java.[†] (Unfortunately, not even ORO can get around the extra escaping required to get regex backslashes and double quotes into Java string literals.)

Even substitutions can be simple. An example from Chapter 2 to "commaify" a number (☞ 67) looks like this in Perl:

```
$text =~ s/(\d)(?=(\d\d\d)+(?!\d))/$1,/g;
```

and this with ORO:

```
text = engine.substitute("s/(\\d)(?=(\\d\\d\\d)+(?!\\d))/$1,/g", text);
```

Traditionally, regular-expression use in Java has a class model that involves pre-compiling the regex to some kind of pattern object, and then using that object later when you actually need to apply the regex. The separation is for efficiency, so that repeated uses of a regex doesn't have to suffer the repeated costs of compiling each time.

So, how does `Perl5Util`, with its procedural approach of accepting the raw regex each time, stay reasonably efficient? It *caches* the results of the compile, keeping a behind-the-scenes mapping between a string and the resulting regex object. (See "Compile caching in the procedural approach" in Chapter 6 ☞ 243.)

It's not perfectly efficient, as the argument string must be parsed for the regex delimiters and modifiers each time, so there's some extra overhead, but the caching keeps it reasonable for casual use.

A Mini `Perl5Util` Reference

The ORO suite of text-processing tools at first seems complex because of the raw number of classes and interfaces. Although the documentation is well-written, it's hard to know exactly where to start. The `Perl5Util` part of the documentation, however, is fairly self-contained, so it's the only thing you really need at first. The next sections briefly go over the main methods.

`Perl5Util` basics — initiating a match

match(*expression,* *target***)**

> Given a match *expression* in Perl notation, and a *target* string, returns true if the regex can match somewhere in the string:
>
> ```
> if (engine.match("/^Subject: (.*)/im", emailMessageText))
> {
>     ⋮
> ```
>
> As with Perl, you can pick your own delimiters, but unlike Perl, the leading `m` is not required, and ORO does not support nested delimiters (e.g., `m{⋯}`).

[†] One further step, I think, would be to remove the Perl trappings and just have separate arguments for the regex and modifier. The whole `m/⋯/` bit may be convenient for those coming to Java from a Perl background, but it doesn't seem "natural" in Java.

Modifier letters may be placed after the closing delimiter. The modifiers allowed are:

- **i** (case-insensitive match ☞ 109)
- **x** (free-spacing and comments mode ☞ 110)
- **s** (dot-matches-all ☞ 110)
- **m** (enhanced line anchor mode ☞ 111)

If there's a match, the various methods described in the next section are available for querying additional information about the match.

substitute(*expression,* *target***)**

Given a string showing a Perl-like substitute *expression*, apply it to the *target* text, returning a possibly-modified copy:

```
headerLine = engine.substitute("s/\\b(Re:\\s*)*//i", headerLine);
```

The modifiers mentioned for `match` can be placed after the final delimiter, as can **g**, which has the substitution continue after the first match, applying the regex to the rest of the string in looking for subsequent matches to replace.[†]

The substitution part of the *expression* is interpreted specially. Instances of `$1`, `$2`, etc. are replaced by the associated text matched by the first, second, etc., set of capturing parentheses. `$0` and `$&` are replaced with the entire matched text. `\U⋯\E` and `\L⋯\E` cause the text between to be converted to upper- and lowercase, respectively, while `\u` and `\l` cause just the next character to be converted. Unicode case conversion is supported.

Here's an example that turns words in all caps to leading-caps:

```
phrase = engine.substitute("s/\\b([A-Z])([A-Z]+)\\b/$1\\L$2\\E/g", phrase);
```

(In Perl this would be better written as `s/\b([A-Z]+)\b/\L\u$1\E/g`, but ORO currently doesn't support the combination of `\L⋯\E` with `\u` or `\l`.)

substitute(*result,* *expression,* *target***)**

This version of the `substitute` method writes the possibly-modified version of the *target* string into a `StringBuffer` result, and returns the number of replacements actually done.

split(*collection,* *expression,* *target,* *limit***)**

The `m/⋯/` *expression* (formatted in the same way as for the `match` method) is applied to the *target* string, filling *collection* with the text separated by matches. There is no return value.

The *collection* should be an object implementing the `java.util.Collection` interface, such as `java.util.ArrayList` or `java.util.Vector`.

---

† An **o** modifier is also supported. It's not particularly useful, so I don't cover it in this book, but it's important to note that it is completely unrelated to Perl's `/o` modifier.

The *limit* argument, which is optional, limits the number of times the regex is applied to *limit* minus one. When the regex has no capturing parentheses, this limits the returned collection to at most *limit* elements.

For example, if your input is a string of values separated by simple commas, perhaps with spaces before or after, and you want to isolate just the first two values, you would use a *limit* of three:

```
java.util.ArrayList list = new java.util.ArrayList();
engine.split(list, "m/\\s+ , \\s+/x", input, 3);
```

An input string of `"USA, NY, NYC, Bronx"`, result in a list of three elements, 'USA', 'NY', and 'NYC, Bronx'. Because you want just the first two, you could then eliminate the "everything else" third element.

An omitted *limit* allows all matches to happen, as does a non-positive one.

If the regex has capturing parentheses, *additional* elements associated with each $1, $2, etc., may be inserted for each successful regex application. With ORO's split, they are inserted only if not empty (e.g., empty elements are not created from capturing parentheses.) Also, note that the *limit* limits the number of regex applications, not the number of elements returned, which is dependent upon the number of matches, as well as the number of capturing parentheses that actually capture text.

Perl's `split` operator has a number of somewhat odd rules as to when it returns leading and trailing empty elements that might result from matches at the beginning and end of the string (☞ 323). As of Version 2.0.6, ORO does not support these, but there is talk among the developers of doing so in a future release.

Here's a simple little program that's convenient for testing `split`:

```
import org.apache.oro.text.perl.Perl5Util;
import java.util.*;

public class OroSplitTest {
    public static void main(String[] args) {
        Perl5Util engine = new Perl5Util();
        List list = new ArrayList();
        engine.split(list, args[0], args[1], Integer.parseInt(args[2]));
        System.out.println(list);
    }
}
```

The `println` call shows each element within [···], separated by commas. Here are a few examples:

```
% java OroSplitTest  '/\./'  '209.204.146.22'  -1
[209, 204, 146, 22]
% java OroSplitTest  '/\./'  '209.204.146.22'  2
[209, 204.146.22]
% java OroSplitTest  'm|/+|' '/usr/local/bin//java'  -1
[, usr, local, bin, java]
% java OroSplitTest  'm/(?=(?:\d\d\d)+$)/'  1234567890  -1
[1, 234, 567, 890]
% java OroSplitTest  'm/\s*<BR>\s*/i'  'this<br>that<BR>other'  -1
[this, that, other]
% java OroSplitTest  'm/\s*(<BR>)\s*/i'  'this<br>that<BR>other' -1
[this, <br>, that, <BR>, other]
```

Note that with most shells, you don't need to double the backslashes if you use single quotes to delimit the arguments, as you do when entering the same expressions as Java string literals.

Perl5Util basics—inspecting the results of a match

The following `Perl5Util` methods are available to report on the most recent successful match of a regular expression (an unsuccessful attempt does not reset these). They throw `NullPointerException` if called when there hasn't yet been a successful match.

group(_num_**)**

Returns the text matched by the _num_th set of capturing parentheses, or by the whole match if _num_ is zero. Returns `null` if there aren't at least _num_ sets of capturing parentheses, or if the named set did not participate in the match.

toString()

Returns the text matched—the same as `group(0)`.

length()

Returns the length of the text matched—the same as `group(0).length()`.

beginOffset(_num_**)**

Returns the number of characters from the start of the target string to the start of the text returned by `group(`_num_`)`. Returns `-1` in cases where `group(`_num_`)` returns `null`.

endOffset(_num_**)**

Returns the number of characters from the start of the target string to the first character after the text returned by `group(`_num_`)`. Returns `-1` in cases where `group(`_num_`)` returns `null`.

groups()

Returns the number of capturing groups in the regex, plus one (the extra is to account for the virtual group zero of the entire match). All *num* values to the methods just mentioned must be less than this number.

getMatch()

Returns an `org.apache.oro.text.regex.MatchResult` object, which has all the result-querying methods listed so far. It's convenient when you want to save the results of the latest `match` beyond the next use of the `Perl5Util` object. `getMatch()` is valid only after a successful `match`, and not after a `substitute` or `split`.

preMatch()

Returns the part of the target string before (to the left of) the match.

postMatch()

Returns the part of the target string after (to the right of) the match.

Using ORO's Underlying Classes

If you need to do things that `Perl5Util` doesn't allow, but still want to use ORO, you'll need to use the underlying classes (the "vast, modular framework") directly. As an example, here's an ORO version of the CSV-processing script on page 386.

First, we need these 11 classes:

```
import org.apache.oro.text.regex.PatternCompiler;
import org.apache.oro.text.regex.Perl5Compiler;
import org.apache.oro.text.regex.Pattern;
import org.apache.oro.text.regex.PatternMatcher;
import org.apache.oro.text.regex.Perl5Matcher;
import org.apache.oro.text.regex.MatchResult;
import org.apache.oro.text.regex.Substitution;
import org.apache.oro.text.regex.Util;
import org.apache.oro.text.regex.Perl5Substitution;
import org.apache.oro.text.regex.PatternMatcherInput;
import org.apache.oro.text.regex.MalformedPatternException;
```

Then, we prepare the regex engine—this is needed just once per thread:

```
PatternCompiler compiler = new Perl5Compiler();
PatternMatcher  matcher  = new Perl5Matcher();
```

Now we declare the variables for our two regexes, and also initialize an object representing the replacement text for when we change '""' to '"':

```
Pattern rCSVmain  = null;
Pattern rCSVquote = null;
// When rCSVquote matches, we'll want to replace with one double quote:
Substitution sCSVquote = new Perl5Substitution("\"");
```

Now we create the regex objects. The raw ORO classes require pattern exceptions to always be caught or thrown, even though we know the hand-constructed regex will always work (well, after we've tested it once to make sure we've typed it correctly).

```
try {
   rCSVmain = compiler.compile(
        "  (?:^|,)                                 \n"+
        "  (?:                                      \n"+
        "      # Either a double-quoted field...    \n"+
        "      \" # field's opening quote           \n"+
        "      (  [^\"]* (?: \"\" [^\"]* )*  )      \n"+
        "      \" # field's closing quote           \n"+
        "   # ... or ...                            \n"+
        "   |                                       \n"+
        "      # ... some non-quote/non-comma text ... \n"+
        "      ( [^\",]* )                          \n"+
        "   )                                       \n",
        Perl5Compiler.EXTENDED_MASK);
   rCSVquote = compiler.compile("\"\"");
}
catch (MalformedPatternException e) {
   System.err.println("Error parsing regular expression.");
   System.err.println("Error: " + e.getMessage());
   System.exit(1);
}
```

ORO's ⌜\G⌟ doesn't work properly (at least as of Version 2.0.6), so I've removed it. You'll recall from the original discussion in Chapter 5 (☞ 216) that ⌜\G⌟ had been used as a precaution, and wasn't strictly required, so it's okay to remove here.

Finally, this snippet actually does the processing:

```
PatternMatcherInput inputObj = new PatternMatcherInput(inputCSVtext);
while ( matcher.contains(inputObj, rCSVmain) )
{
    String field; // We'll fill this in with $1 or $2
    String second = matcher.getMatch().group(2);
    if ( second != null ) {
        field = second;
    } else {
        field = matcher.getMatch().group(1);
        // If $1, must replace paired double quotes with one double quote
        field = Util.substitute(matcher,         // the matcher to use
                                rCSVquote,       // the pattern to match with it
                                sCSVquote,       // the replacement to be done
                                field,           // the target string
                                Util.SUBSTITUTE_ALL); // do all replacements
    }
    // We can now work with the field . . .
    System.out.println("Field [" + field + "]");
}
```

Phew! Seeing all that's involved certainly helps you to appreciate `Perl5Util`!