Java Regular Expressions

Regular Expressions Overview

Suppose you needed to find a specific IPv4 address in a bunch of files? This is easy to do; you just specify the IP address as a string and do a search. But, what if you didn’t know in advance which IP address you were looking for, only that you **wanted to see all IP addresses in those files**?

Even if you could, you wouldn’t want to specify every possible IP address to some searching tool! You need a way to specify “all IP addresses” in a compact form. That is, you want to tell your searching tool to show anything that matches *number.number.number.number*.

This is the sort of task we use REs for. *Regular expressions* (or “regexp” or “regex” or RE) are a way to specify concisely a group of text strings. You can specify a *pattern* (RE) for phone numbers, dates, credit-card numbers, email addresses, URLs, and so on. You can then use searching tools to find text that matches.

Depending on the tool that supports REs, you can use them to match text (*does the text match the RE? true or false*), extract matching text, or perform match and replacement operations.

Originally, REs were limited in what sort of strings they could represent. Over time, folks wanted more expressive REs, and new features were added. In the 1980s, original REs became standardized by POSIX as *basic REs (BREs*), and later, *extended REs* (*EREs*) were standardized with more features (and fewer backslashes needed). Since then, people have kept adding more features to the utilities and languages that supported REs. Today, there are many RE dialects. Java has one of the most expressive and powerful dialects.

Regular expressions are often used with sanitizing, validating (discussed in Java II), and data parsing tasks. **Every programmer needs to know about regular expressions; they are used in most programming languages, and in nearly any program that processes text data.** See `java.util.regex.Pattern` for more information; you can also practice with `RegExLab.jar` found in our class resources. REs are used in many parts of Java, including `String.match`, `Scanner`, etc.

The Good Enough Principle

With REs, the concept of “good enough” applies. Consider the pattern used above for IP addresses. It will match any valid IP address, but also strings that look like 7654321.300.0.777 or 5.3.8.12.9.6 (possibly an SNMP OID). Crafting an RE to match only valid IPv4 addresses is possible, but rarely worth the effort. It is unlikely your search of system configuration files would find such strings, and if a few turn up you can easily eye-ball them to determine if they are valid IP addresses.

It is usually possible to craft more precise REs. But in real life, you only need an RE good enough for your purpose at hand. If a few extra matches are caught, you can usually deal with them. (Of course, using REs in global search and replace commands, you will need to be more precise or you may end up changing more than you intended.)

An RE is a pattern, or template, against which strings can be matched. Either strings match the pattern or they don't. If they do, the actual text that matched parts of the RE can be saved in named variables (sometimes called *registers*). These can be used later, to either match more text, or to transform the matching string into a related string.

Pattern matching for text turns out to be one of the most useful and common operations to perform on text. Over the years, a large number of tools have been created that use REs, including most text editors and world processors, command line utilities such as Unix (and Linux) `grep`, and others. Even using wildcards to match file names at the command line could be considered a type of RE.

While the idea of REs is standard, tools developed since POSIX EREs use slightly different syntaxes. Perl's REs are about the most complex and useful dialect, and are sometimes referred to as *PREs* or *PCREs* (Perl Compatible Regular Expressions). Java supports a rich syntax for REs, described in java.util.regex.Pattern. (A hopefully more readable summary is available on our class resources.)

Most RE dialects work this way: some text (usually one line) is read. Next, the RE is matched against it. In a programming environment such as Perl or when using an editor, if the RE matches then some additional steps (such as modification of the line) may be done. With a tool such as Java's `Scanner`, the part of the text that matched is just returned.

> Top-down explanation (from POSIX `regex(7)` man page):
>
> An RE is one or more *branches* separated with "`|`" and matches text if any of the branches match the text. A branch is one or *pieces* concatenated together, and matches if the 1st piece matches, then the next matches from the end of the first match, until all pieces have matched. A piece is an *atom* optionally followed by a modifier: "`*`", "`+`", "`?`", or a ***bound***. An *atom* is a single character RE or "`(`*`RE`*`)`".
>
> If that makes sense to you, you're a smarter person than I.

I believe the easiest way to understand REs is through examples. The basic syntax (that is the same for any dialect including Java's) is:

any char matches that char.

. (a dot) matches (almost) any one character. It may or may not match EOL, depending if the DOTALL option is set. Also, it won't match invalid multi-byte sequences.

\char matches *char* literally if *char* is a meta-char (such as "`.`" or "`*`"). This is called ***escaping*** (so "\\." is read as *escape period*). Never end a RE with a single "`\`".

In Java, some special sequences are defined, such as "`\n`" to mean a newline, "\\t" for a tab, "`\\`" to mean a literal backslash, "`\`*`digits`*" to mean a Unicode character, and so on. Java has other special sequences that start with a backslash. (For example, "`\p{Lu}`" for any Unicode uppercase letter.) For this reason, you should avoid escaping non-meta characters; while legal, you might accidentally be typing a special sequence.

Java supports a convenient form of *quoting* that escapes all enclosed characters. With: "`\Q`*`your-text-here`*`\E`", all characters between the `\Q` and `\E` are taken literally.

> You need to be careful with backslashes! This is because REs are stored in `String`s, and that means backslashes are interpreted twice: once as a `String`, and once as a regular expression. So if you wanted an RE of “`\.`”, you need the `String` `"\\."`, and if you wanted “`\\`”, you would need a `String` of `"\\\\"`.

[*list*] called a ***character class***, matches any one character in *list*. The *list* can include one or more ranges, such as `a-z`, `A-Z`, `0-9`, or a sub-range of those. The *list* can include multiple ranges or a mix of ranges and characters. In Java, you can use any part of Unicode in a range, such as “`[\u3040-\u309F]`” to match Hiragana characters.) An example: “`[a-zA-Z0-9_$]`” would match the characters that are legal in a Java variable name. (There are predefined classes too, some shown below.)

[^*list*] any character not in *list*.

To include a literal “`]`” in the list, make it the first character (following a possible **`^`**) or you can escape it (“`\]`”); escaping is the only legal way in some other languages such as JavaScript. To include a literal “**`-`**”, make it the first (following a possible **`^`**) or last character.

Metacharacters other than “`\`” and “]” lose their meaning in a list. (So you don’t need a backslash for a dot or open brace, as in this ugly example of a list of four characters: “`[].[-]`”. With escaping: “`[.[\]-]`”.)

List can include one or more predefined lists, or *character classes*. In some RE dialects including Java, there are several predefined ranges you can use. Most are denoted with a backslash+*character*, such as “`\d`” to denote a predefined list of all decimal digits. The “`.`” is a predefined character class that matches any one character.

POSIX REs have some predefined ranges that work only for ASCII: **`alnum`**, **`digit`**, **`punct`**, **`alpha`**, **`graph`** (same as **`print`**, except space), **`space`** (any whitespace), **`blank`** (space or tab only), **`lower`**, **`upper`**, **`cntrl`**, **`print`** (any printable char), or **`xdigit`**. An example use (using POSIX notation): “`[[:upper:][:digit:]]`”. Java does not support this notation.

In Java, you can use these classes using backslash notation: “`\p{Lower}`”, “`\p{Print}`”, etc., and some non-POSIX ones including “`\p{ASCII}`”, “`\p{javaWhitespace}`”, `\R` (end of line character(s)), etc. Some of the more useful ones: `\d` (digit), `\D` (non-digit), `\s` (whitespace), and `\S` (non-space). So you could define a list of uppercase letters and digits as “`[\p{Upper}\p{Digit}]`” or as “`[\p{Upper}\d]`”. (Yes, you can include lists within lists.)

Note that a dot does not normally match newlines. In some cases, you can pass a flag to the pattern compiler to change that, but this works too (and in other languages’ REs): “`[\s\S]`”. (Any complementary pair will work.)

> Java supports additional character class (list) features, not discussed here.

In addition to single character REs, **REs can match a sequence of characters.** *Concatenated REs* means matching a *string* of text that matches each RE in turn. (So “`A.B`” matches three characters starting with `A` and ending with `B`.)

Quantifiers (Repetition)**:**

`RE{count}` Exactly *count* of *RE*. (Example: "`A.{10}B`" matches 12 characters that start with `A` and end with `B`.)

`RE{min,max}` `max` (but not the comma) can be omitted for infinity. (Example: "`X{3,}`" means 3 or more `X`es.)

`RE*` zero or more of RE. (Same as "`RE{0,}`".)

`RE+` one or more of RE. (Same as "`RE{1,}`".)

`RE?` zero or one of RE. (Same as "`RE{0,1}`".)

> These are all ***greedy*** quantifiers. Greedy and non-greedy quantifiers are explained below.

`RE1|RE2` either match RE1 or match RE2.

`(RE)` A grouped RE, matches RE. For example, "`(ab){3}`" would match "`ababab`".

Boundaries: POSIX REs don't define word *delimiters*, known as *boundaries* or ***anchors***. These match the null (zero-length) string at the beginning and/or end of something, such as a word. Java supports several boundaries, including these (first two are in POSIX):

`^RE` an RE *anchored* to the beginning of the line.

`RE$` an RE *anchored* to the end of the line.

`\b` a word boundary (for example, "`\b12\b`" matches "`I have 12 dollars`", but not "`I have 123 dollars`").

`\A` beginning of data boundary.

`\z` end of data boundary. ("`\Z`" is the same but doesn't include the final line terminator.)

Note that "`^A`" matches `A` at the beginning of a line, but "`X^A`" matches three characters; in this case the "^" is taken literally. The exact rules for this behavior depend on what options are in effect. In *multiline* mode, "`foo\Rbar`" would match `foo` at the end of one line and `bar` at the beginning of the next. The "`\R`" matches the line terminator in between. (Some non-Java dialects use "`$^`" to match a line terminator.) Multiline mode has other effects too.

Precedence Rules:

1. Repetition ("*****"and "**+**") takes precedence over concatenation, which in turn takes precedence over alternation ("**|**"). So "`ab|cd`" means either "`ab`" or "`cd`". A whole sub-expression may be enclosed in parentheses to override these precedence rules (ex: "`a(b|c)d`").
2. In the event that a given RE could match more than one substring of a given string, th**e RE matches the one starting earliest in the string.**
3. If an RE could match more than one substring starting at the same point, it **matches the longest**. (This is often called ***greedy* matching** and is the default mode.) Sub-expressions in an RE also match the longest possible substrings, **subject to the constraint that the whole match be as long as possible**.

The +, *, ?, and {min,max}, as noted above match the longest possible match (rule 3); they are greedy quantifiers.

> The greedy match of the beginning part of an RE would prevent the following part from matching anything, in many cases. In this event, ***backtracking*** occurs, and a shorter match is tried for the first part. For example, the 5 parts of the RE **`xx*xx*x`** (matched against a long string of '`x`'es) will end up matching this way: `x|xxxxx|x||x`. (*Show how.*)

Backtracking is powerful, but sometimes unnecessary. Since it is a performance hog, Java supports maximum match (greedy) quantifiers that don't backtrack. They are called the **possessive quantifiers**: `?+`, `*+`, `++`, etc. These grab a maximum amount of text but never give it back.

Java supports **reluctant** (or non-greedy; I like the term *generous*. :) quantifiers as well. These match the shortest possible amount that allows the overall RE to still match. They look the same as the greedy ones, but with a "`?`" appended: `+?`, `*?`, `??`, and `{min,max}?`.

Back reference The actual text that matched each group is remembered in a *register*, numbered by counting open parenthesis. For example, the regular expression: "`([0-9]*)\1`" would match "`123123`" or "`11`", but not "`12312`". The precise definition is: a '`\`' followed by a non-zero decimal digit *d* (a back reference) matches the same sequence of characters matched by the *d*th parenthesized sub-expression (numbering sub-expressions by the positions of their opening parentheses, left to right). For example: "`([bc])\1`" matches "`bb`" or "`cc`", but not "`bc`". For example: **`(((ab)c(de)f))`** has **`\1=abcdef`, `\2=abcdef`, `\3=ab`, `\4=de`**

> For performance reasons, Java supports non-capturing grouping as well.

Examples of Simple Regular Expressions

`abcdef` Matches "`abcdef`".

`a*b` Matches zero or more "`a`"s followed by a single "`b`". For example, "`b`" or "`aaaaab`".

`a?b` Matches "`b`" or "`ab`".

`a+b+` Matches one or more "`a`"s followed by one or more "`b`"s: "`ab`" is the shortest possible match; others are "`aaaab`" or "`abbbbb`" or "`aaaaaabbbbbb`".

`.*` and `.+` These two both match all the characters in a string; however, the first matches every string (including the empty string), while the second matches only strings containing at least one character.

`^public static void main\s*\(.*\)` This matches a string that starts with "`main`" followed by, optional whitespace, then an opening and closing parenthesis with optional stuff between.

`^#` This matches a string beginning with "`#`".

`\\$` This matches a string ending with a single backslash. The regex contains two backslashes for escaping.

`\$` This matches a single dollar sign, because it is escaped.

`[a-zA-Z0-9]` Any ASCII letter or digit.

`[^ tab]+` (Here *tab* stands for a single tab character.) This matches a string of one or more characters, none of which is a space or a tab. Usually this means a word. (Simpler would be “`\H+`” for non-horizontal space.)

`^(.*)\n\1$` This matches a string consisting of two equal substrings separated by a newline.

`.{9}A$` This matches any nine characters followed by an “`A`”, at the end of the line.

`^.{15}A` This matches a string that starts with 16 characters, the last of which is an “`A`”.

`^$` Matches (empty) strings. (“`.`” would be an RE that matches non-empty strings.)

`foo\Rbar` Matches two lines, the first ending with foo and the second starting with bar. (Only in `MULTILINE` mode.)

`(.*)\1` Matches any string that repeats, e.g., “`abcabc`” or “`byebye`”.

`^(.*)\R\1$` Find duplicate lines that are next to each other. (Only in `MULTILINE` mode.)

Example: *Parse a text configuration file, skipping comment and blank lines:*

`^($|\h*#)    //` *Either an empty line, or one that starts with ‘#’*

Example: *Determine if some string is an integer (number):*

This is an interesting problem because it comes up a lot. Assuming you need to allow whole numbers with an optional leading plus or minus sign. Also assume you don’t need to support weird forms of integers such as `1e6`, `12.`, `1,000,000`, or `0x1A`. Also assume you don’t care if `-0` or `+0` is considered a valid integer (you can modify the code if you do care).

```
^[+-]?[0-9]+$
```

Example: *Find only* ***legal*** *IPv4 addresses of “num.num.num.num”, where “num” is at least one digit, at most three digits including optional leading zeros, and in the range of 0..255 (note: all one line; split here for readability):*

```
\b(([01]?\d?\d|2[0-4]\d|25[0-5])\.){3}
   ([01]?\d?\d|2[0-4]\d|25[0-5])\b
```

However, this simpler RE may be “good enough”:

```
\b(\d{1,3}\.){3}\d{1,3}\b
```

Example: *Match* Roman Numerals *(note: all one line; split here for readability):*

```
\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})
(IX|IV|V?I{0,3})\b
```

> Newlines and regular expressions don’t get along well. POSIX REs don’t support them. Some RE engines can use “multiline” matching (including Java). Many RE engines support a number of options like this.

Email addresses as defined by RFC-822 and the newer RFC-5322 standards were not designed to be regexp-friendly. In some cases, only modern (non-POSIX) regular expression languages can handle them. Compare these two email address matching REs: ex-parrot.com and stackoverflow.com. Keeping in mind the *good-enough* principle, here’s the much shorter one I use to validate the email addresses in my RSS feeds (this is not the Java RE dialect; see if you can follow it anyway):

```
([a-zA-Z0-9_\-])([a-zA-Z0-9_\-\.]*)@(\[((25[0-5]|2[0-4][0-
9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\.){3}|((([a-zA-Z0-9\-]
+)\.)+))([a-zA-Z]{2,}|(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-
9][0-9]|[0-9])\])(  *\([^\)]*\) *)*
```

As can be seen from that last example, REs can be very complex. This makes them hard to read or debug. You can often break up a complex RE into multiple REs, each matching a portion of the string, to make each piece simpler and easier to debug and to test.

For long REs, it can be a good idea to format them across multiple lines, with spaces added for readability. Comments can be included as well within Java REs. The following nonsense example was ~~stolen~~ adopted from Leif Moldskred, posted in the `comp.lang.java.programmer` newsgroup ("G#" means "grouping number, which can be use with back references later):

```
Pattern exampleRegex =
  Pattern.compile( "^\\s*" // Start of string, optional whitespace
    + "(19[5-9][0-9]|20[0-9][0-9])" // G1: year from 1950 to 2099
    + "-(0[1-9]|1[012])"      // '-' divisor, G2: month, 01 to 12
    + ":((\\w+\\s?)*)"        // ':' divisor, G3: one or more words,
                              //     separated by a single whitespace
    + "(:(.+)?)$"             // Optional: ':' divisor
                              // G6: Remaining contents of line
  );
```

Since Java 8, a *checker framework* is available (but not included) that include many new *annotations*. These instruct the compiler to perform additional checks for possible errors. One of these new annotations is **`@Regex`**. This provides compile-time verification that a `String` intended to be used as a regular expression is a properly formatted regular expression. (It doesn't check that the RE does what you want, however.)

Using Regular Expressions in Java

The simplest way to use REs is with the **`String.matches`** method:

```
if ( "foobar".matches(".*foo.*") )
    System.out.println( "It matches!" );
```

This is a whole-string match; to match just a part of the string, you need the ".*" on either side.

To do more complex matching, use the classes in `java.util.regex` package. In Java, you pass an RE as an argument to either the `Pattern.compile(`*`RE`*`)` or the `Pattern.compile(`*`RE`*`, `*`flags`*`)` static methods. To use that RE (some `Pattern` object), create a `Matcher` object from it with some text to match against. The flags alter how the matcher works; for example, case-insensitive matching or multi-line mode flags can be set.

At this point, the matcher hasn't tried to match anything yet. You need to invoke a method to attempt a match: **`matches`** is a whole-string match, **`find`** matches the regexp anywhere within the string. These methods return a `boolean` to indicate success or failure. If the match succeeded, you can then use other `Matcher` methods to extract the matched text (including capture group text), do search and replace operations, etc.

Here’s a couple of examples of matching an RE against a whole `String` (useful for data validation):

```
Pattern p = Pattern.compile( "a*b" );
Matcher m = p.matcher( "aaaaab" );
boolean b = m.matches();
```

```
// Or just use the simpler:
```

```
boolean b = Pattern.matches( "a*b", "aaaaab" );
```

```
// Or use the String.matches method:
```

```
boolean b = "aaaaab".matches( "a*b" );
```

```
// Or chain method calls:
```

```
boolean b = Pattern.compile( "a*b" )
    .matcher( "aaaaab" ).matches();
```

```
boolean b - Pattern.compile("foo\\Rbar",
Pattern.MULTILINE).matcher("foo\nbar").matches();
```

`Matcher` objects have many useful methods:

`matches()` compares the whole string to the RE (i.e., there’s an implied "`^`" and "`$`" around the RE). (Also, the `lookingAt` method has just an implied "`^`".)

`find()` has no implied anchors, so it can find substrings. You can use `find()` in a loop to find the first, second, etc. match. Each `find` call pick up from where the last one left off, that is, at the first character not matched by the previous match (unless you `reset` the matcher).

> If the string is large, you don’t have to search the whole string. Matcher allows you to specify the start and end of a *search region*. The searching methods only match against that region.

`group()` returns the `String` that matched. When using capturing groups, you can return the *n*th group with **`group(`*`n`*`)`**. You can find the index of the start and end of each group (from the source string) using **`start(`*`n`*`)`** and **`end(`*`n`*`)`**.

To replace one group of the matched text you can use the `start(`*`n`*`)` and `end(`*`n`*`)` methods to get the index of the start and end of the matching *n*th group. You can then use simple `String` or `StringBuilder` methods to replace that. If you want to find and replace just the first match, or all matches, instead of some specific match, just use the methods **`replaceFirst()`** and **`replaceAll()`**. For example, the following replaces all `foo`s with `bar`s:

```
    String data = "A foo in the hand is worth"
     + "two foos in the bush";
    Matcher m = Pattern.compile("foo").matcher(data);
    System.out.println( m.replaceAll("bar") );
```

The `String.replaceAll` method works the same way. (*Show Matcher methods, RegEx.java.*)

To replace just the second “foo” with “bar”:

```
// version 1
StringBuilder sb = new StringBuilder(
   "A foo in the hand is worth two foos"
   + "in the bush");
Matcher m = Pattern.compile("foo").matcher(sb);
m.find(); m.find();
sb.replace(m.start(), m.end(), "bar");

// version 2
...
Matcher m = Pattern.compile("foo.*(foo)").matcher(sb);
m.find();
sb.replace(m.start(1), m.end(1), "bar");
```

Most regular expression engines (including Java’s) are not secure, in that malicious users can craft inputs that cause RE engines to stack overflow or backtrack so much it can take minutes to run some RE, causing a server backlog and eventual crash. (This has actually happened.) Google developed RE2, a safe RE package with some rarely-used features (such as back-references) that require backtracking omitted. RE2 does no backtracking or recursion.