[Skip to content](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html" \l "container)

* [*Safari*](https://www.safaribooksonline.com/home/)
* [Recommended](https://www.safaribooksonline.com/r/)
* [Queue](https://www.safaribooksonline.com/s/)
* [Search](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html)
* [Expand Nav](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html)
  + [Recent](https://www.safaribooksonline.com/recent/)
  + [Topics](https://www.safaribooksonline.com/t/)
  + [Highlights](https://www.safaribooksonline.com/u/p253751/)
  + [Settings](https://www.safaribooksonline.com/u/)
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)
  + [Settings](https://www.safaribooksonline.com/u/)**10** days left in your trial. [Subscribe](https://www.safaribooksonline.com/subscribe/).
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)

[Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html)

* [Search in book...](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html)

Top of Form

Bottom of Form

* [Toggle Font Controls](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html)
* [Share this](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html)
  + [Twitter](https://twitter.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html&text=Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource&via=safari)
  + [Facebook](https://www.facebook.com/sharer/sharer.php?u=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Google Plus](https://plus.google.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Email](mailto:?subject=Safari:%20Chapter%204.%20XML%20and%20Rational%20Functional%20Tester&body=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html%0D%0Afrom%20Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource%0D%0A)

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.html)

[Appendix A. Advanced Logging Techniques](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/index.html)

[Index](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/index.html)

**Appendix B. Regular Expressions in Rational Functional Tester**

**Jeffrey R. Bocarsly**

*Regular expression pattern matching provides a powerful way to match and parse strings. This power is brought to bear in Rational Functional Tester in the areas of object recognition (pattern matching recognition property values in the Object Map) or in Verification Points (pattern matching to captured data values). The history between Rational Functional Tester and regular expressions is a bit complicated, because the original (Java-only) version of the tool was released before Java had a standard regular expressions package. Because of this, IBM Rational included its own regular expression class based on the Apache package* org.apache.regexp*. Through multiple releases of the tool (including two re-brandings), the regular expressions feature remains based on the Apache implementation, as it is with Rational Functional Tester currently (*com.rational.test.util.regex.Regex*). For this reason, the Apache-based Rational Functional Tester package is discussed in this appendix, and not the Java package. If you are interested, you can find the underlying Apache regular expression package documentation at* [*http://jakarta.apache.org/regexp*](http://jakarta.apache.org/regexp)*.*

*When Rational released the VB.NET flavor of Rational Functional Tester, it included regular expression classes that maintain parallel syntax with the Java flavor (*Rational.Test.Util.Regex*), so the discussion here applies to both the Java and the VB.NET flavors of Rational Functional Tester.*

*In addition to using regular expressions for object recognition and in Verification Points, the Rational Functional Tester regular expressions package can be used in your scripts for matching test data (as can the Java and .Net regular expressions packages). Because there is a large amount of information about the Java and VB.NET packages available in print and on the Internet, this appendix covers only the Rational Functional Tester package.*

**Using Regular Expressions in Rational Functional Tester Scripts**

You may have noticed that Rational Functional Tester actually has two classes to deal with regular expressions, Regex and RegularExpression. The RegularExpression class simply wraps a regular-expression pattern, and is used for interactions with Rational Functional Tester methods, such as Verification Point methods and TestObject's find() method, which use the second class, Regex, to evaluate matches. The Regex class is a lightweight regular expression engine that actually performs matches and parses text. Because Regex is the actual engine, our discussion here is limited to it.

Regex can be used for string processing in a script. To access the Rational Functional Tester regular expression class in a script, you need to create a regular expression object. In Java, this looks like:

Regex re = new Regex("pattern");

In VB.NET, the analogous code is

Dim re As Regex = New Regex("pattern")

The pattern argument to the Regex constructor is a String containing a regular expression that describes a text pattern that you want to match. When you create your regular expression object, you “compile” a regular expression pattern. The pattern string may be any valid regular expression pattern.

The Regex class has two main methods that enable you to test for a pattern match and extract out the matched pattern: matches() and getMatch(). The matches() method takes a String as its argument, and returns a boolean evaluating whether the String matches the compiled regular expression pattern. A simple Java example is:

Image

In this case, matches() returns true because the superset String argument “abcdef” matches the subset pattern “def”. The same example in VB.NET is:

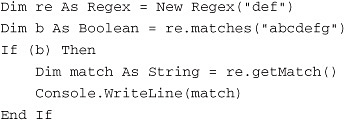
Image

The method getMatch() returns the portion of the argument String that matches the pattern. Because getMatch() returns a String based on a previous match, the matches() method must be called before getMatch(). A Java example of how this works is:

Image

Image

In this example, the variable match is set to the value “def”. If matches() has not been called before getMatch(), a null reference is returned (no exception is thrown). A null value is also returned if there is no match. The example in VB.NET is:



As noted earlier, in the Java flavor of Rational Functional Tester, the power of regular expressions was offered before they were a standard part of the Java libraries. Therefore, Rational Functional Tester uses the Regex class, a customized version of the Apache Regular Expression implementation instead of the standard Java regular expressions library. You are still free to use the Java Regular Expressions engine java.util.regex (released in Java 1.4) in any Java code you write in your scripts. Likewise, you are free to use System.Text.RegularExpressions in VB.NET to parse your string data. The Rational Functional Tester API also contains two other regular expressions wrapper classes, ApacheRegExp and JavaRegExp, each of which provides an interface similar to that of the Regex class.

You can access the full Apache package within Rational Functional Tester (org.apache.regexp) simply by importing com.rational.test.util.regex.internal.\* into your Java scripts; however, this does not appear to be exposed in VB.NET.

**Writing Regular Expressions**

Regular expression patterns are built from a rich library of *operators* (also called *metacharacters*) that can be used to construct elegant patterns. The key to successfully writing the pattern that you actually want is *following the pattern matching rules precisely*. It is quite a simple matter to write a pattern that gives unexpected results, because it can be a challenge to see the implications of your operator syntax on a match, especially when you use multiple operators. Paying rigorous attention to each operator’s rules will help you avoid writing a pattern that results in unintended consequences. Key regular expression rules include:

• Each regular expression operator represents a single rule

• A pattern consists of a single character, unless it is explicitly declared to have multiple characters.

• Operators can be combined by placing one operator to the immediate right of another.

• Operators are processed from left-to-right, subject to normal constraints imposed by parentheses.

• A superset string matches a subset pattern.

Note

Although the broad outlines of how regular expression operators work are fairly standardized, there are differences between different implementations. If you are familiar with regular expressions from outside the Java context, or simply from a different implementation of regular expressions in Java, there might be some differences here that you will have to adjust to.

As noted previously, Regular Expression patterns are built out of common pieces of text along with operators that specify a text pattern to match. Operators are typically indicated by punctuation characters. The operators considered in this discussion are:

• \* (asterisk)—Matches 0 or more of the previous pattern

• + (plus)—Matches 1 or more of the previous pattern

• ( ) (parentheses)—Defines pattern groups

• . (period)—Matches any individual character

• ? (question mark)—Matches 0 or 1 of the previous pattern

• | (pipe)—A logical Or

• [ ] (square brackets)—Defines a “character class”

• { } (curly braces)—Matches a pattern a defined number of times

This appendix looks at each of these operators in the context of specific examples that illustrate how the operators work, and in addition, how they can interact with each other (with apologies to Lewis Carroll).

**Non-Operator Characters**

*Match themselves*

As noted in [Chapter 9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09), “[Advanced Rational Functional Tester Object Map Topics](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09),” many regular expressions don’t need any operators to be useful. If you need to match the beginning of a piece of text, but not its end, you can simply use the leading characters in the text for your pattern:

Image

The period in the preceding examples is actually an operator but ignore that for now. (See the following for a discussion of the period.)

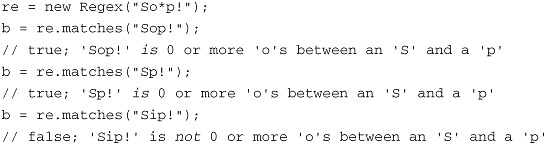
**The Asterisk Operator: \***

*Matches 0 or more of the previous pattern*

In the pattern “So\*p!”, the asterisk (\*) operator modifies the previous pattern, which is the character ‘o’. Any string that has zero or more ‘o’s that trail an ‘S’ and lead a ‘p’ matches:

Image

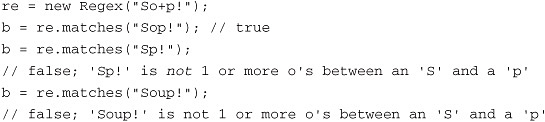
Note that the second example does not qualify—it does not have ‘zero or more ‘o’s trailing an ‘S’ and leading a ‘p’, because there is a ‘u’ in the position before the ‘p’. As you’ll appreciate from the following examples, a literal application of the rules is demanded:



**The Plus Operator: +**

*Matches 1 or more of the previous pattern*

The plus (+) operator is identical to the \*, except that a minimum of one instance of the pattern is required for a match.



**The Parentheses Operator: ( )**

*Defines pattern groups*

Thus far, you have seen how an operator can modify the previous pattern when the previous pattern is a single character (see the previous examples). You can start to create complex patterns by defining groups of characters as a single pattern. Enclosing multiple characters in parentheses causes the regular expression engine to treat them as a group. Any operator following the group applies to the entire group of characters. Note how the parentheses affect the regular expression processing in the following examples:

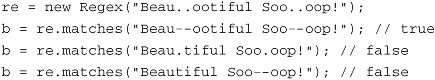
Image

Image

**The Period Operator: .**

*Matches any single character (except newline)*

The period (.) operator matches one character of any sort, except a newline character (in Windows, a newline is actually indicated by two characters, a carriage-return followed by a linefeed). Zero characters do not match.



As indicated, operators can be combined to operate on each other: for example, following a ‘.’ with a ‘\*’ (meaning the pattern ‘.\*’) means ‘zero or more repetitions of any character’, or, in other words: match any number of any characters. If you have used wildcards (such as DOS wildcards), the pattern ‘.\*’ is roughly equivalent to the wildcard ‘\*’.

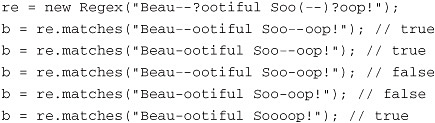
Image

Note that the ‘.\*’ operator combination is particularly potent—there is nothing it won’t match. Use it with care.

**The Question Mark Operator: ?**

*Matches 0 or 1 of the previous pattern*

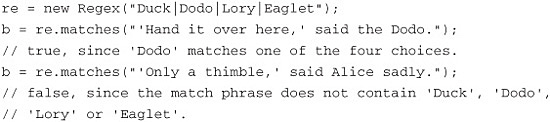
The question mark (?) operator limits the previous pattern to either 0 or 1 occurrences. In the examples that follow, you can see how the ? works with grouped and ungrouped patterns.



**The Pipe Operator: |**

*A logical Or*

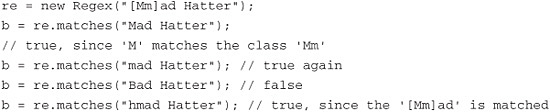
Sometimes, it is easiest to match to a set of discrete values rather than try to write a complex pattern that matches words that are logically related, but are not related in their spellings. Common examples include colors, currencies, or city, state, province, and country names. The pipe (|) operator provides the ability match one of a set of possible values.



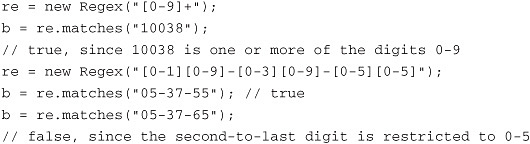
**The Square Brackets Operator: [ ]**

*Defines character classes*

A character class defines a set of characters, one of which matches to the specified position in a string. The specified position is the position of the square brackets in the pattern. So, if you place a character class at the beginning of a pattern, this class matches *only* to the first character in the match string. If you place it in the second position, it provides a set of characters that can match at the second position, and so on. In the following snippets, character matches to either ‘M’ or ‘m’ in the first position are accepted.



Ranges of characters can be denoted in a character class by separating the beginning and ending characters of the range with a dash (-). Ranges include numeric range ([0-9]), and alpha ranges ([a-z], [A-Z]). Multiple ranges can be included in a single class. For example, [a-zA-Z] matches any alpha character regardless of case.



Character classes can exclude characters at a given position. This is done by placing the hat operator ^ at the beginning of the class. So, if you want to indicate which characters are *not* allowed to match at a position, you can write a pattern like the following pattern, which *excludes* the numbers 2 through 9 for the first position:

Image

**The Curly Braces Operator: { }**

*Matches a pattern a defined number of times*

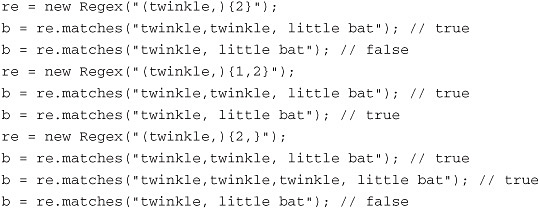
The curly braces create complex patterns that follow these rules:

• x{n}—Match x exactly n times

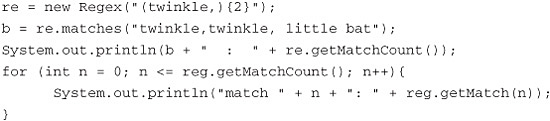
• x{n,}—Match x at least n times

• x{n,m}—Match x at least n times and not more than m times

If you have repeated subpatterns (such as repeated words or symbols) in a pattern, you can use the curly braces to specify either the number of repeats of the subpattern, or the upper and lower limits for the number of repeats.



Note that the curly braces modify the pattern behavior, not the number of *matched groups*. If you check the value returned by getMatchCount() in the following example, Regex reports a value of 1:



The code above returns the following matches. The zeroth match, which is retrieved by passing a 0 argument to getMatch(), always returns the complete match. Match 1 returns the contents that match the pattern in the first pair of parentheses:

Image

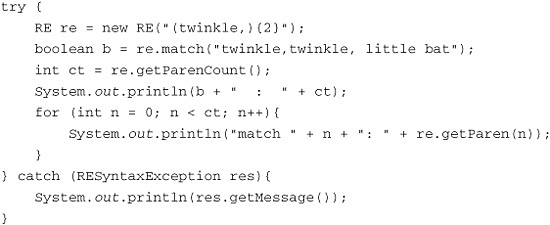
Now, if you need a pattern that returns two match groups, you can simply double your pattern instead of using the curly brace operator:

Image

This pattern with a match count of 2 returns two matches, one to each group, in addition to the zeroth match:

Image

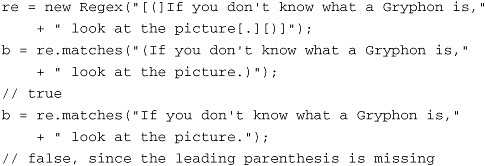
However, you may want to use the curly brace syntax but also access to each match individually. To do this, you can use the underlying Apache implementation (as noted previously) based on the RE class. The following example shows how, with the Apache implementation, the curly braces produce three matches ("twinkle,twinkle,", "twinkle,", and "twinkle,"):



This code produces the output identical to the previous two-group pattern using the curly brace syntax.

**Escaping Regular Expression Operators**

Operators are escaped with square brackets (what you are actually doing is taking advantage of the fact that character classes treat the regular expression metacharacters as regular characters) or with a backslash. In the following example, the parentheses and the period are escaped:



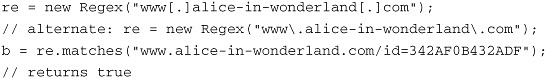
You can escape operators with the backslash; an equivalent pattern to that in the previous example is:

\(If you don't know what a Gryphon is, look at the picture\.\)

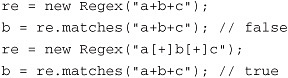
In Java, backslashes in Strings must themselves be escaped, so the Java String for this pattern is:

"\\(If you don't know what a Gryphon is, look at the picture\\.\\)"

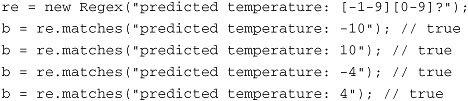
In the first example in this appendix, a pattern was shown that matches a URL. In that example, we left the periods in the URL unmodified and ignored the fact that the period is an operator. Because the period matches itself just like it matches any other single character, this pattern won’t fail to match despite the fact that the period operator is not properly escaped. Properly escaping the pattern provides a more robust pattern that won’t accidentally create an unintended match:



Mathematical formulas can be treated using this approach:



A common character that is useful to use in a character class is the dash (-). Normally, the dash is a regular character in a regular expression; however, the dash is an operator in a character class (it indicates a range; see the previous character class discussion). To use a dash in a character class as a normal character, it must be listed *first* in the class list (where it can’t define a range). The regular expression engine will understand this and will treat the dash as a normal member of the character class:



By using the character class [-1-9], either a dash (used as a negative sign) or a number in the range 1–9 is accepted as the first character in the temperature field.

**Escaped Characters**

Regular expression patterns can use the regular “C-type” escaped characters, such as \r for carriage-return, \n for newline, and \t for the tab character. In addition, you can use the following escaped characters to define a pattern:

• \w—Matches any word character

• \W—Matches any nonword character

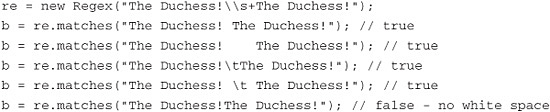
• \s—Matches any whitespace character

• \S—Matches any non-whitespace character

• \d—Matches any decimal digit

• \D—Matches any nondigit

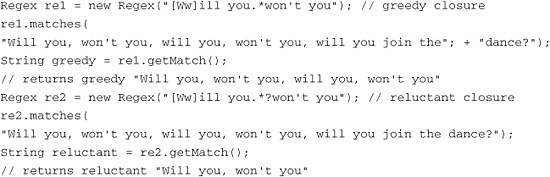
These can be used as in the following, where any white space is accepted between the two phrases, including spaces and tabs:



**Greedy Versus Reluctant Closure**

Regular expression packages have a feature known as closure, which refers to the manner in which the matching algorithm performs its task: Does the regular expressions engine look for the largest possible match, or the smallest possible match? A trivial example of this would be using the pattern j.\*zz to match the string jazzjazzjazz. The maximal match would be to the whole string jazzjazzjazz. The minimal match would be only the first jazz sequence. If the algorithm looks for the maximal match, then it is termed *greedy closure*. When it looks for the minimal match, it is called *reluctant closure*.

Greedy or reluctant closure applies to a subset of operators called *closure operators*. These are +, \*, ?, and {m,n}; however, reluctancy of {m,n} is not currently supported. The default closure behavior for all these is greedy closure. Reluctant closure is indicated by appending a trailing ? after the closure operator to which it applies. The greedy form of the pattern above, j.\*zz, has the reluctant form j.\*?zz. You can implement either greedy or reluctant closure as follows:



**Regex Games with the Queen of Hearts**

To increase your comfort level with regular expressions, we show some additional examples in this section that illustrate some of the subtle ways the regular expressions engine interprets patterns. This first example sets the baseline by matching a lowercase alpha followed by a space and a name. The expected result 'f Hearts' is matched.

Regex re = new Regex("[a-z] Hearts");  
re.matches("Queen of Hearts"); // matches 'f Hearts'

In the second example, you match an *uppercase* alpha followed by a space and Hearts. No match occurs because the character immediately before the space is lowercase.

Regex re = new Regex("[A-Z] Hearts");  
re.matches("Queen of Hearts"); // false

Things get more interesting when you try to combine an uppercase and lowercase specification with the pipe operator (logical or). All that is matched is the single character 'u.' The reason for this is that the operators are processed left-to-right, so the regular expression engine reads this request as “find a lowercase character OR an uppercase character followed by a space and of Hearts.” As the previous example shows, there is no uppercase character followed by a space and of Hearts. Therefore, the request is satisfied only by the first clause, the first lowercase character found, which is a 'u' in the word Queen:

Image

To satisfy yourself that this is the correct interpretation, group the second clause after the pipe in parentheses. When you do this, you get the same return value, 'u', confirming that Regex reads the operators left-to-right.

Image

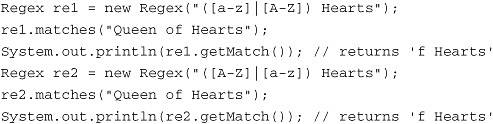
You can further verify this by switching the order of the character classes. In the two previous examples, the lowercase class appears first. In the following, the lowercase class is second, and the uppercase class is first. The return value is the expected uppercase 'Q', because ‘the first uppercase character’ satisfies the request, processing from left to right.

Image

You can confirm this again by grouping the second clause with parentheses:

Image

If you group the 'or' clause in parentheses, the order of the character classes becomes irrelevant because the request is for the first lowercase *or* uppercase character followed by a space and of Hearts. This is satisfied either way by 'f Heart'.



**Putting It All Together**

Now, you are in a position to examine some robust examples. Although the typical regular expression that you might need is quite simple, periodically there are cases where a more elaborate pattern can solve a problem elegantly without forcing you to write a great deal of code. We look at a couple of examples here: handling common URLs with regular expressions, parsing HTML with regular expressions, and parsing data from strings with regular expressions.

**Handling URLs with Regular Expressions**

Let’s say you have a web application development project, and instances of the application are deployed on multiple environments (for example, QA, staging, and production environments). You want a single suite of Rational Functional Tester regression scripts to run against all three environments, instead of having to maintain three sets of scripts (one for each environment). Assume that your URLs look like the following:

• <http://www1.alice-in-wonderland.com/hatters?mad=true>

• <http://www2.alice-in-wonderland.com/hatters?mad=false>

• <http://www3.alice-in-wonderland.com/hatters?mad=true>

These URLs have three possible hosts (denoted by www1, www2, and www3), and a parameter is passed in the URL (“mad”) with different possible values.

As with the previous URL examples (see the previous examples in this appendix and those in [Chapter 9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09)), character classes are useful for pattern matching to URLs. For a first attempt, you can use the approach of the previous example in this appendix and simply match the host portion of the URL:

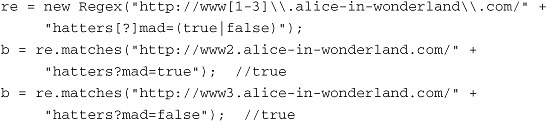
Image

If you want to write an even more specific pattern, you can include matches to the parameter and the allowed values that can be passed for the parameter:

Image

Image

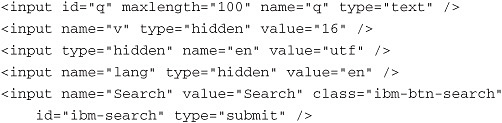
When you execute this match, the matches() method returns false, so something must be wrong. On reviewing the regular expression, note that standard URL syntax uses a question mark to indicate that parameters follow, but the question mark has its own meaning in regular expressions syntax—it matches 0 or 1 of the previous pattern (an ‘s’, in this case). Therefore, the match fails not because the target string lacks 0 or 1 ‘s’ characters (it has an ‘s’ in the proper position), but because it has the question mark as an element to be matched, and this is not represented in the pattern. To make this pattern work, the question mark has to be escaped:



This pattern correctly matches not only to the different hosts in your URL, but also to the true/false parameter that is passed in the URL.

**Parsing HTML with Regular Expressions**

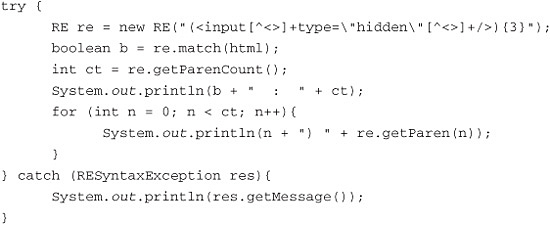
Almost any data found in HTML can be handled in Rational Functional Tester by the usual routes. However, you might have to parse HTML in your Rational Functional Tester code in a couple of situations to complete a test. One of these situations is where the HTML contains hidden tags that hold data that needs to be captured. Because these tags are not rendered, they often cannot be captured by the standard approaches. In this case, you can capture the HTML containing the hidden tags (often with a call to getProperty()), and then you can parse the tags and the data they contain with regular expressions. Consider the following HTML:



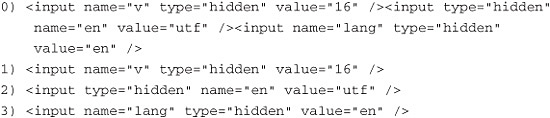
If you want to pull the hidden tags out, you could write a pattern to match just those tags that contain the attribute type="hidden". You would need to specify other characters that are permitted inside a tag to match the rest of the tag contents. In this case, it is easier to write a pattern to match what is *not* allowed in a tag because far fewer characters are not allowed inside a tag than are allowed. This can be done with a negated character class:

(<input[^<>]+type="hidden"[^<>]+/>)

With this pattern, the tag’s angle brackets are required for a match, along with the tag name (<input>), and the hidden type attribute. The pattern allows any characters that are not angle brackets, which are the main characters not allowed inside a tag. This pattern matches any of the individual hidden tags in the HTML; in order to pull all the tags out, you need access to the full curly brace operator, which also means using the RE class, not the Regex class as Regex only matches the first <input> tags as a group, while RE will give access to each tag as a separate match. In Java, this looks like:



Note that you have to modify the pattern slightly by appending the {3} to the end of the pattern to match the three hidden tags in the HTML. The double quotes also need to be escaped with a backslash, as is required by Java. The output from this match is:

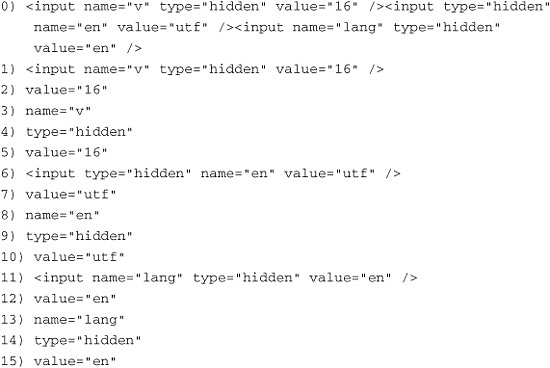


If you wanted to attack the problem at an even deeper level, you can modify the pattern so that the individual tag attributes are themselves grouped as matches. To build an HTML attribute-based pattern, consider the characteristics of the target HTML. First, all the syntactic entities in each tag are separated by spaces. Second, each <input> tag has the same set of attributes (name, type, and value) but they are not in a constant order. So, to write a more elaborated pattern, you have to account for spaces and handle the attribute order.

The space problem is simply solved—you can place a leading or trailing space in the pattern for each attribute (while accounting for the space after the tag name and before the end of the tag). The attribute order problem is also fairly easily solved—you can use the pipe operator to match any of the three attributes. The resulting pattern is:

(<input ((name="[\\w]+" )|(type="hidden" )|(value="[\\w]+" ))+/>){3}

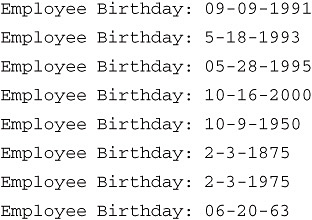
In this pattern, you match the attribute *values* with \w (any word character), and you use a trailing space in each of your attribute patterns. You nest the attribute patterns in a pair of parentheses, so that you can apply the + operator to the group. This enables RE to find all three attributes. Finally, the pattern needs a trailing space after the tag name. The output for this looks like:



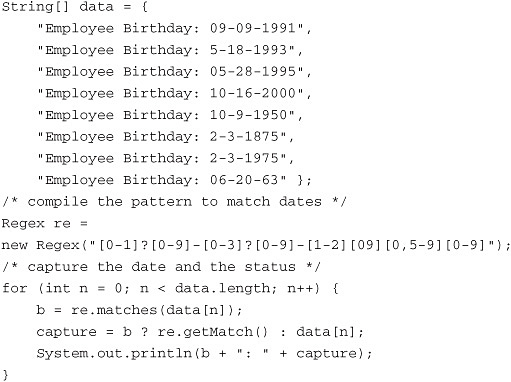
The full output has 16 lines, counting match 0, which always shows the full match. Note also that the value attribute is repeated in each group; this happens because submatches are defined by parentheses, and the pattern uses nested attribute subpatterns in order to apply the + operator to the group of attribute subpatterns.

**Parsing Data with Regular Expressions**

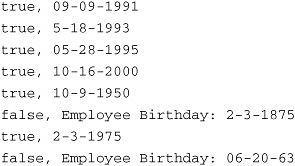
Imagine you have a series of strings that are captured from a target application of the form “Employee birthday: mm-dd-yyyy.” However, because some of the data is quite old, the dates have been input in varying formats. The formatting rules that the date data must follow and that you must validate are the month and day fields can have either one or two digits, but the years should all be corrected to four digits. In addition, the years must be later than 1950. Sample data looks like:



Your task is to validate all the rules and to capture the actual date from each valid string, or the whole string if it is invalid. You can use a regular expression to test compliance with the rules, and then write a bunch of string-handling code to pull out the dates. Or, you can take advantage of the getMatch() method and simplify your approach. In Java, this looks like:



The output from this example is:



**Regular Expression Control Flags**

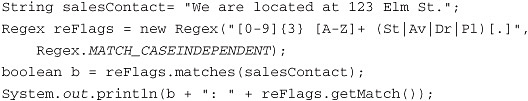
The final aspect to Regex that you examine is the class control flags. In Rational Functional Tester, three flags can be passed to the Regex constructor:

• Regex.MATCH\_NORMAL—Case-sensitive match, the default

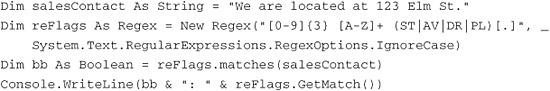
• Regex.MATCH\_CASEINDEPENDENT—Case-insensitive match

• Regex.MATCH\_MULTILINE—Match newlines

The following Java example shows a case-insensitive match to the street type designation (Street, Avenue, Drive, or Place). Note that the pattern is written with capitals, but the flag allows lowercase matches to be made.



Finally, in the VB.NET version, System.Text.RegularExpressions.RegexOptions constants are used, not Regex constants.



**Further Reading**

Friedl, J. *Mastering Regular Expressions*, Third Edition. Sebastopol, CA: O’Reilly Media, Inc., 2006.

For a tutorial on Java’s regular expressions package java.util.regex, see <http://java.sun.com/docs/books/tutorial/essential/regex/>.

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.html)

[Appendix A. Advanced Logging Techniques](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/index.html)

[Index](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/index.html)

**Welcome to Safari**. Remember, your free trial will end on March 7, 2015, but you can [subscribe at any time](https://www.safaribooksonline.com/subscribe/)

* [Recommended](https://www.safaribooksonline.com/r/)
* [Queue](https://www.safaribooksonline.com/s/)
* [Recent](https://www.safaribooksonline.com/recent/)
* [Topics](https://www.safaribooksonline.com/t/)
* [Settings](https://www.safaribooksonline.com/u/)
* [Blog](http://blog.safaribooksonline.com/)
* [Support](http://support.safaribooksonline.com/?prod=flow)
* [Support](http://msupport.safaribooksonline.com/?prod=flow)
* [Feedback](mailto:feedback@safaribooksonline.com)
* [Sign Out](https://www.safaribooksonline.com/accounts/logout/)

© 2015 [Safari](http://www.safaribooksonline.com/).   
[Terms of Service](https://www.safaribooksonline.com/terms/) / [Membership Agreement](https://www.safaribooksonline.com/membership-agreement/) / [Privacy Policy](https://www.safaribooksonline.com/privacy/)

[**Make font larger Make font smaller**](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html)