Article
Michael Broesdorf · Jul 22, 2016 16m read

# Using Regular Expressions in Caché

## 1.About this article

Just like Caché pattern matching, Regular Expressions can be used in Caché to identify patterns in text data – only with a much higher expressive power. This article provides a brief introduction into Regular Expressions and what you can do with it in Caché. The information provided herein is based on various sources, most notably the book " Mastering Regular Expressions " by Jeffrey Friedl and of course the Caché online documentation. The article is not intended to discuss all the possibilities and details of regular expressions. Please refer to the information sources listed in chapter 5 if you would like to learn more. If you prefer to read off-line you can also download the PDF version of this article.

Text processing using patterns can sometimes become complex. When dealing with regular expressions, we typically have several kinds of entities: the text we are searching for patterns, the pattern itself (the regular expression) and the matches (the parts of the text that match the pattern). To make it easy to distinguish between these entities, the following conventions are used throughout this document:

Text samples are printed in a monospace typeface on separate, without additional quotes:

```
This is a "text string" in which we want to find "something".
```

Unless unambiguous, regular expressions within the text body are visualized with a gray background such as in this example: `\".*?\"`.

Matches are highlighted in different colors when needed:

```
This is a "text string" in which we want to find "something".
```

Larger code samples are printed in boxes like in the following example:

```
set t="This is a ""text string"" in which we want to find ""something""."
set r="\"".*?\"""
w $locate(t,r,,,tMatch)
```

## 2.Some history (and some trivia)

In the early 1940s, neuro-physiologists developed models for the human nervous system. Some years later, a mathematician described these models with an algebra he called " regular sets " . The notation for this algebra was named " regular expressions " .

In 1965, regular expressions are for the first time mentioned in the context of computers. With qed, an editor that was part of the UNIX operating system, regular expressions start to spread. Later versions of that editor provide a command sequence g/regular expression/p (*global, regular expression, print*) that searches for matches of the regular expression in all text lines and outputs the results. This command sequence eventually became the stand-alone UNIX command line program " grep " .

Today, various implementations of regular expressions (RegEx) exist for many programming languages (see section 3.3).

# 3.Regex 101

Just like Caché pattern matching, regular expressions can be used to identify patterns in text data – only with a much higher expressive power. The following sections outline the components of regular expressions, their evaluation and some of the available engines, details of how to use this are then described in chapter 4.

## 3.1.Components of regular expressions

### 3.1.1.Regex meta characters

The following characters have a special meaning in regular expressions.

```
.  *  +  ?  (  )  [  ]  \  ^ $  |
```

If you need to use them as literals, you need to escape it using the backslash. You can also explicitly specify literal sequences using `\Q <literal sequence> \E`.

### 3.1.2.Literals

Normal text and escaped characters are treated as literals, e.g.:

| | |
|---|---|
| abc | abc |
| \f | form feed |
| \n | line feed |
| \r | carriage return |
| \v | (vertical) tab |
| \0+three digits (e.g. `\0101`) | Octal number<br>The regex engine used in Caché (ICU) supports octal numbers up to \0377 (255 in decimal system). When you migrate regular expressions from another engine make sure you understand how it handles octal numbers. |
| \x+two digits (e.g. `\x41`) | Hexadecimal number<br>The ICU library does provide more options of handling hex numbers, please refer to the ICU documentation (links can be found in section 5.8) |

### 3.1.3.Anchors

With anchors, you match positions in your text/string, e.g.:

- \A Start of string

- \Z End of string
- ^ start of text or line
- $ end of text or line
- \b Word boundary
- \B Not word boundary
- \< Start of word
- \> End of word

Some RegEx engines behave differently, for instance by the definition of what exactly constitutes a word and which characters are considered word delimiters.

## 3.1.4.Quantifiers

With quantifiers, you can specify how often the preceding element may occur to be a match:

- {x} exactly x occurrences
- {x,y} minimum x, maximum y occurrences
- * 0 or more; equivalent to {0,}
- + 1 or more; equivalent to {1,}
- ? 0 or 1

### Greediness

Quantifiers are *‘greedy,’* they grab as many characters as possible. Suppose we have the following text string and want to find text in quotes:

```
This is "a text" with "four quotes".
```

Because of the greedy nature of the selectors, the regular expression `\".*\"` would find too much text:

```
This is "a text" with "four quotes".
```

In this example, the regular expression `.*` tries to capture as many characters as possible that are between a pair of quotes. However, because the dot selector ( `.` ) does also match quotes we do not get the result we want.

With some regex engines (including the one used by Caché) you can control the greediness of the quantifiers by adding a question mark to it. So the regular expression `\".*?\"` now matches the two parts of the text that is in quotes - exactly what we are looking for:

```
This is "a text" with "four quotes".
```

## 3.1.5.Character classes (ranges)

Square brackets are used to specify ranges or sets of characters, e.g. `[a-zA-Z0-9]` or `[abcd]` – in regex speech this is referred to as a character class. A range matches single characters, so the order of characters within the range definition is irrelevant – `[dbac]` returns the same matches as `[abcd]`.

To exclude a range of characters simply put ^ in front of the range definition (within the square brackets!): `[^abc]` matches anything except a, b or c.

Some regex engines do provide pre-defined character classes (POSIX), e.g.:

- `[:alnum:]` `[a-zA-z0-9]`
- `[:alpha:]` `[a-zA-Z]`

- `[:blank:]`    `[ \t]`
- …

### 3.1.6.Groups

Parts of a regular expression can be grouped using a pairs of parenthesis. This can be useful to apply quantifiers to a group of selectors as well as for referencing the groups from both within the same regex (back references) as well as from the Caché Object Script code calling the regular expression (capture buffers). Groups can be nested.

The following regex matches strings consisting of a three-digit number, followed by a dash, followed by 3 pairs of an uppercase letter and a digit, followed by a dash, followed by the same three-digit number as in the first part:

`([0-9]{3})-([A-Z][[0-9]){3}-\1`

This example shows how you can use *back references* (see below) to match not only structure but also content: the back reference (purple) tells the engine to look for the same three-digit number at the end as at the beginning (yellow). It also demonstrates how to apply a quantifier to more complex structures (green).

The regex from above would match on the following string:

`123-D1E2F3-123`

It would not match on these:

`123-D1E2F3-456`    (the last three-digits are different from the first three)

`123-1DE2F3-123`    (the middle part does not consist of three letter/digit-pairs)

`123-D1E2-123`    (the middle part contains only two letter/digit-pairs)

Groups will also populate the so-called capture buffers (see section 4.5.1). This is a very powerful feature that allows you to match and extract information at the same time!

### 3.1.7.Alternations

Use the pipe character to specify alternations, e.g. `skyfall|done`. This allows for matching more complex expressions as the character classes described in section 3.1.5.

### 3.1.8.Back references

Back references allow you to refer to previously defined groups (selectors in parenthesis). The following example shows a regular expression that matches three consecutive characters that have to be equal:

`([a-zA-Z])\1\1`

Back references are specified by \x, whereas x represents the x-th bracketed expression.

### 3.1.9.Rules of precedence

1. [] before ()
2. , + and ? before sequence: `ab` equates to `a(b*)`, not `(ab)*`
3. Sequence before alternation: `ab|c` equates to `(ab)|c`, not `a(b|c)`

## 3.2.Some theory

The evaluation of regular expressions is usually implemented with one of the following two methods (the descriptions are simplified, please refer to the literature mentioned in chapter 5 for in-depth discussions):

1. Text-driven (DFA – *Deterministic Finite Automaton*)
   The engine steps through the input text character by character and tries to match what it has so far. When it actually reaches the end of the input text, it declares success.

2. Regex-driven (NFA – *Non-deterministic Finite Automaton*)
   The engine steps through the regular expression token by token and tries to apply it to the text. When it actually reaches (and matches) the last token, it declares success.

Method 1 is deterministic, execution time depends on the length of the input text only. The order of the selectors in the regular expression does not impact execution time.

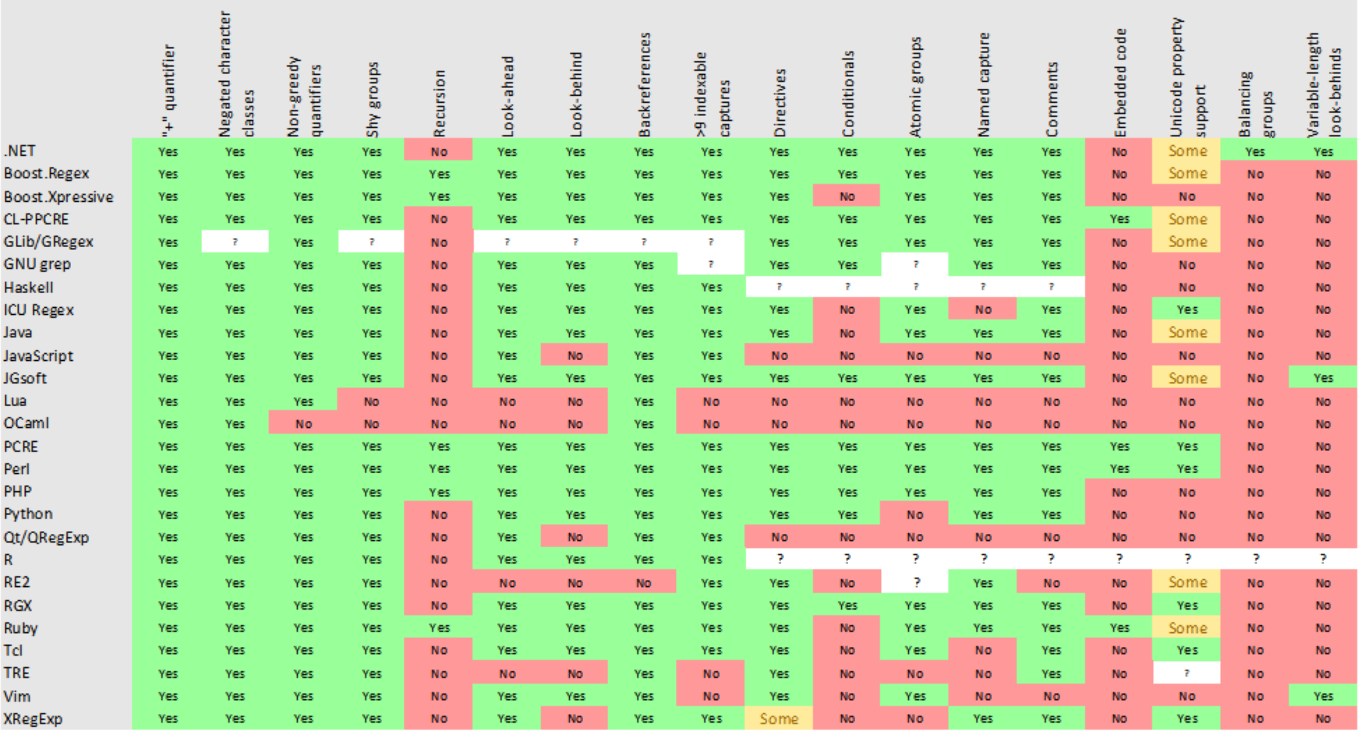
Method 2 is non-deterministic, the engine plays through all combinations of the selectors in the regular expression until it finds a match or runs into an error. Hence, this method is particularly slow when it does *not* find a match (because it has to play through all possible combinations). The order of the selectors *does* have an impact on execution time. However, this method allows for backtracking and capture buffers.

## 3.3.Engines

There are many different regex engines available, some are a built-in part of programming languages or operating systems, others being libraries that can be used almost anywhere. Here are some regex engines, grouped by evaluation method:

- DFA: grep, awk, lex
- NFA: Perl, Tcl, Python, Emacs, sed, vi, ICU

The folowing table represent a comparison of available regex features in various programming languages and libraries:



|  | "+" quantifier | Negated character classes | Non-greedy quantifiers | Shy groups | Recursion | Look-ahead | Look-behind | Backreferences | >9 indexable captures | Directives | Conditionals | Atomic groups | Named capture | Comments | Embedded code | Unicode property support | Balancing groups | Variable-length look-behinds |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| .NET | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Some | Yes | Yes |
| Boost.Regex | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Some | No | No |
| Boost.Xpressive | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | No | No | No | No |
| CL-PPCRE | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Some | No | No |
| GLib/GRegex | Yes | ? | Yes | ? | No | ? | ? | ? | ? | Yes | Yes | Yes | Yes | Yes | No | Some | No | No |
| GNU grep | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | ? | Yes | Yes | ? | Yes | Yes | No | No | No | No |
| Haskell | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | ? | ? | ? | ? | ? | No | No | No | No |
| ICU Regex | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | No | Yes | No | Yes | No | Yes | No | No |
| Java | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | No | Some | No | No |
| JavaScript | Yes | Yes | Yes | Yes | No | Yes | No | Yes | Yes | No | No | No | No | No | No | No | No | No |
| JGsoft | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Some | No | Yes |
| Lua | Yes | Yes | Yes | No | No | No | No | Yes | No | No | No | No | No | No | No | No | No | No |
| OCaml | Yes | Yes | No | No | No | No | No | Yes | No | No | No | No | No | No | No | No | No | No |
| PCRE | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | No |
| Perl | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | No |
| PHP | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No |
| Python | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | No | No | No | No |
| Qt/QRegExp | Yes | Yes | Yes | Yes | No | Yes | No | Yes | Yes | No | No | No | No | No | No | No | No | No |
| R | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| RE2 | Yes | Yes | Yes | Yes | No | No | No | No | Yes | Yes | No | ? | Yes | No | No | Some | No | No |
| RGX | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | No | No |
| Ruby | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Some | No | No |
| Tcl | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | No | Yes | No | Yes | No | Yes | No | No |
| TRE | Yes | Yes | Yes | Yes | No | No | No | Yes | No | Yes | No | No | No | Yes | No | ? | No | No |
| Vim | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | No | Yes | No | Yes | No | No | No | No | No | Yes |
| XRegExp | Yes | Yes | Yes | Yes | No | Yes | No | Yes | Yes | Some | No | No | Yes | Yes | No | Yes | No | No |

Details can be found here: https://en.wikipedia.org/wiki/Comparisonofregularexpressionengines

# 4.RegEx and Caché

InterSystems Caché uses the ICU library for regular expressions, the Caché online documentation describes many of its features. Please refer to the online documentation of the ICU library for full details (including things like back references etc.) – links to ICU can be found in section 5.8. The following sections are intended to give you a quick introduction on how to use it.

## 4.4.$match() and $locate()

In Caché ObjectScript (COS), the two functions $match() and $locate() provide direct access to most of the regex features provided by the ICU library.$match(String, Regex) searches the input string for the specified Regex pattern. When it finds a match it returns 1, otherwise it returns 0.

Examples:

- w $match("baaacd","`.*(a) \1 \1.*`") returns 1
- w $match("bbaacd","`.*(a) \1 \1.*`") returns 0

$locate(String,Regex,Start,End,Value) searches the input string for specified regex pattern – just like $match(). However, $locate() gives you more control and it returns more information. In Start, you can tell $locate at which position it should start searching for patterns in the input string. When $locate() finds a match, it returns the position of the first character of the match and sets End to the next character position after the match. The content of the match is returned in Value.

If $locate() does not find a match it returns 0 and does not touch the contents of End and Value (if specified).End and Value are passed as references, so be careful if you use it repeatedly (e.g. in loops).

Example:

- w $locate("abcdexyz","`.d.`",1,e,x) returns 3, e is set to 6, x is set to "cde"

$locate() performs the pattern matching and can return the content of the first match at the same time. If the content of all matches needs to be extracted you can call $locate() repeatedly in a loop or you can use the methods provided by %Regex.Matcher (see next section).

## 4.5.%Regex.Matcher

%Regex.Matcher provides access to the regex functionality of the ICU library, just like $match() and $locate(). However, %Regex.Matcher also provides access to some advanced features that make more complex tasks very easy to use. The following sections will revisit capture buffers, look at the possibility to replace strings with regular expressions and ways to control the runtime behavior.

### 4.5.1.Capture buffers

As we have already seen in the sections about groups, back references and $locate(), regular expressions allow you to simultaneously search for patterns in text and return the matched content. This works by putting the parts of the pattern you would like to extract in a pair of parenthesis (grouping). Upon successful matching, the capture buffers contain the contents of all matched groups. Note that this is slightly different from what $locate() provides with its value-parameter: $locate() returns the content of the whole match itself while capture buffers give you access to *parts* of the match (the groups).

To use it, you create an object of the class %Regex.Matcher and pass it the regular expression and the input string. Then you can call the one of the methods provided by %Regex.Matcher to perform the actual work.

Example 1 (simple groups):

```
set m=##class(%Regex.Matcher).%New("(a|b).*(de)", "abcdeabcde")
w m.Locate() returns 1
w m.Group(1) returns a
w m.Group(2) returns de
```

Example 2 (nested groups and back reference):

```
set m=##class(%Regex.Matcher).%New("((a|b).*?(de))(\1)", "abcdeabcde")
w m.Match()             returns      1
w m.GroupCount          returns      4
w m.Group(1)            returns      abcde
w m.Group(2)            returns      a
w m.Group(3)            returns      de
w m.Group(4)            returns      abcde
```

(note the order of the nested groups – because the opening parenthesis marks the beginning of a group, inner groups have a higher index number than the outer ones)

As mentioned before, capture buffers are a very powerful feature as they allow you to match patterns and extract the matched content at the same time. Without regular expressions, you would have to find your matches in step one (e.g. using the pattern match operator) and extract the matched content (or parts of it) based on some criteria in step two.

If you need to group parts of your pattern (for instance to apply a quantifier to that part) but you don ’ t want to populate a capture buffer with content of the matched part, you can define the group as “ non-capturing ” or “ shy ” by putting a question mark, followed by a colon in front of the group as in the example 3 below.

Example 3 (shy group):

```
set m=##class(%Regex.Matcher).%New("((a|b).*?(?:de))(\1)","abcdeabcde")
w m.Match()                   returns     1
w m.Group(1)                  returns     abcde
w m.Group(2)                  returns     a
w m.Group(3)                  returns     abcde
w m.Group(4)                  returns     <REGULAR EXPRESSION>zGroupGet+3^%Regex.Matc
her.1
```

## 4.5.2.Replace

%Regex.Matcher does also provide methods to replace the contents of matches right away: ReplaceAll() and ReplaceFirst():

```
set m=##class(%Regex.Matcher).%New(".c.","abcdeabcde")
w m.ReplaceAll("xxxx")        returns    axxxxeaxxxxe
w m.ReplaceFirst("xxxx")      returns    axxxxeabcde
```

You can also reference groups in your replacement strings. If we add a group to the pattern from the previous example, we can reference its content by including $1 in the replacement string:

```
set m=##class(%Regex.Matcher).%New(".(c).","abcdeabcde")
w m.ReplaceFirst("xx$1xx")    returns    axxcxxeabcde
```

Use $0 to include the full content of the match in the replacement string:

```
w m.ReplaceFirst("xx$0xx")    returns    axxbcdxxeabcde
```

### 4.5.3.OperationLimit

In section 3.2 we learned about the two methods of evaluating a regular expression (DFA and NFA). The regex engine used in Caché is a nondeterministic finite automaton (NFA). Therefore, the duration to evaluate various regular expressions on a given input string may vary. [1]

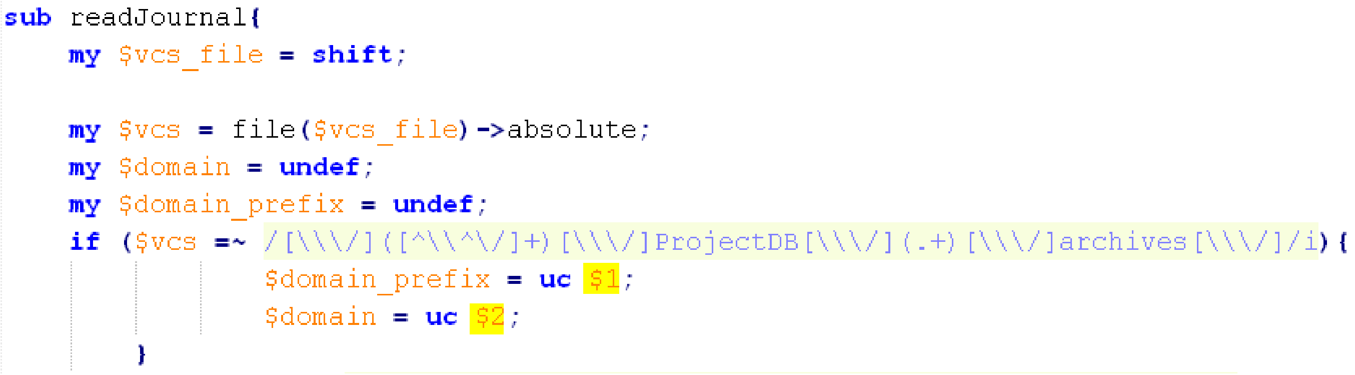
You can use the property OperationLimit of a %Regex.Matcher object to limit the number of execution units (so-called *clusters*). The exact duration to execute a cluster depends on your environment. Typically a duration for the execution of a cluster is very few milliseconds. By default, the OperationLimit is set to 0 (no limit).

## 4.6.Real world example: migration from Perl to Caché

This section describes the regex-related part of a migration from Perl to Caché. The Perl script actually consisted of dozens more or less complex regular expressions which were used to both match and extract content.

If there was no regex functionality available in Caché, the migration project would have turned into a major effort. However, regex functionality is available in Caché, and the regular expressions from the Perl script could be used in Caché almost without any changes.

Here is a part from the Perl script:



```
sub readJournal{
    my $vcs_file = shift;

    my $vcs = file($vcs_file)->absolute;
    my $domain = undef;
    my $domain_prefix = undef;
    if ($vcs =~ /[\\\/]([^\\^\/]+)[\\\/]ProjectDB[\\\/](.+)[\\\/]archives[\\\/]/i){
            $domain_prefix = uc $1;
            $domain = uc $2;
      }
```

The only change to the regular expressions required to move it from Perl to Caché was the /i-modifier (making the regex case insensitive) – this had to be moved from the end of the regex to the beginning.

In Perl, contents of the capture buffers are copied into special variables ($1 and $2 in the Perl code above). Almost all of the regular expression in the Perl project used this mechanism. To resemble this, a simple wrapper method was written in Caché Object Script. It uses %Regex.Matcher to evaluate a regular expression against a text string and returns the contents of the capture buffer as a list ($lb()).

The resulting Caché Object Script code looks like this:

```
if ..RegexMatch(
       tVCSFullName,
       "(?i)[\\\/]([^\\^\/]+)[\\\/]ProjectDB[\\\/](.+)[\\\/]archives[\\\/]",
       .tCaptureBufferList)
       {
             set tDomainPrefix=$zcvt($lg(tCaptureBufferList,1), "U")
             set tDomain=$zcvt($lg(tCaptureBufferList,2), "U")
       }
```

…

```
Classmethod RegexMatch(pString as %String, pRegex as %String, Output pCaptureBuffer="
") {

       #Dim tRetVal as %Boolean=0
       set m=##class(%Regex.Matcher).%New(pRegex,pString)
       while m.Locate() {
             set tRetVal=1
             for i=1:1:m.GroupCount {
                    set pCaptureBuffer=pCaptureBuffer_$lb(m.Group(i))
             }
       }
       quit tRetVal
}
```

# 5.Reference information

## 5.7.General information

General information and tutorials:

- http://www.regular-expressions.info/engine.html

Tutorials and examples:

- http://www.sitepoint.com/demystifying-regex-with-practical-examples/

Comparison of several regex engines:

- https://en.wikipedia.org/wiki/Comparison_of_regular_expression_engines

Cheat sheet:

- https://www.cheatography.com/davechild/cheat-sheets/regular-expressions/pdf/

Book:

- Jeffrey E. F. Friedl: “ Mastering Regular Expressions ” (see http://regex.info/book.html)

## 5.8.Caché Online Documentation

- Overview about the usage of regular expressions in Caché: http://docs.intersystems.com/latest/csp/docbook/ DocBook.UI.Page.cls?KEY=GCOS_regexp
- Documentation of $match():

http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=RCOS$fmatch

- Documentation of $locate(): http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=RCOS$flocate

- Class reference of %Regex.Matcher: http://docs.intersystems.com/latest/csp/documatic/%25CSP.Documatic.cls?APP=1&LIBRARY=%25SYS&CLASSNAME=%25Regex.Matcher

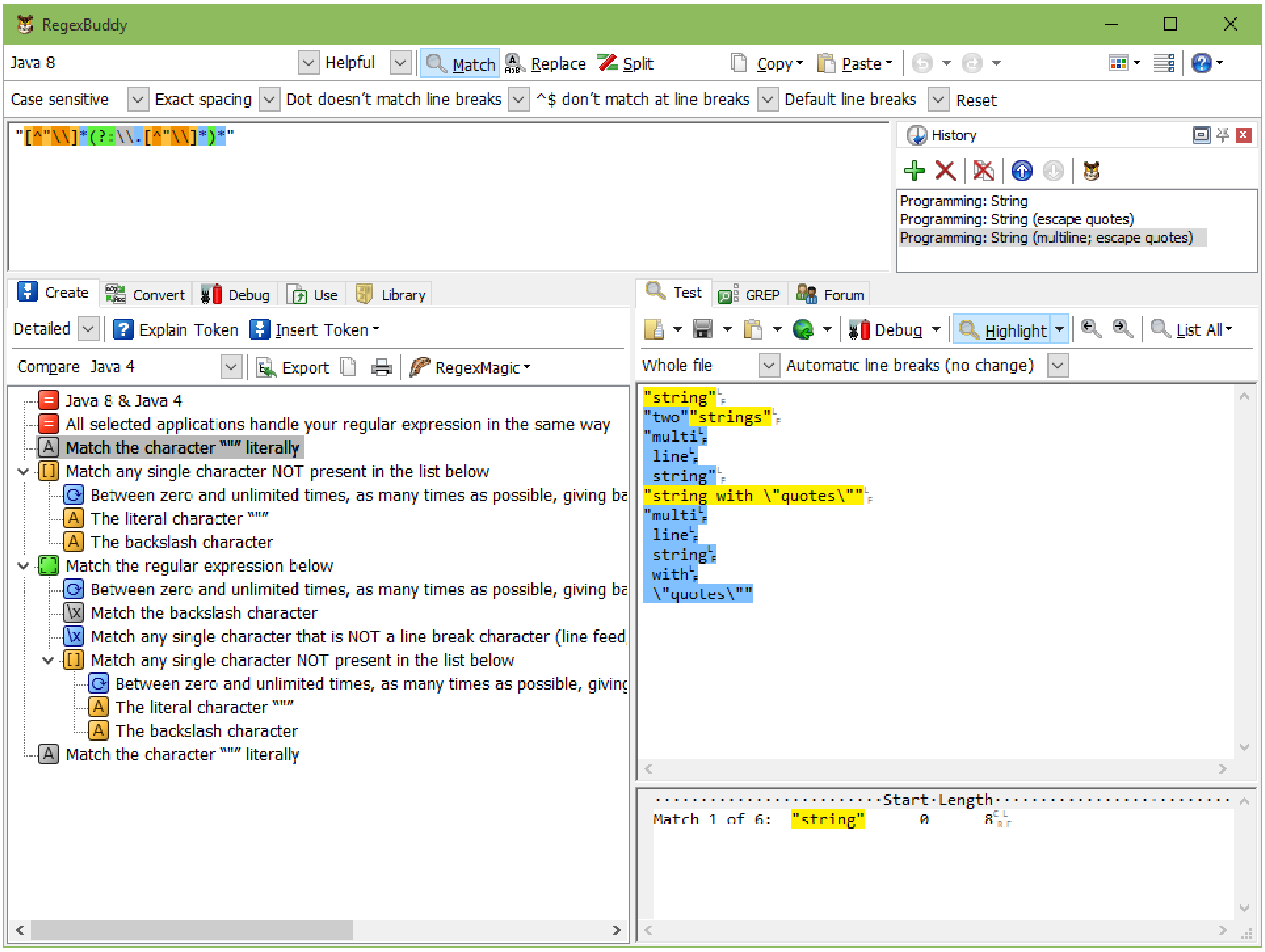
## 5.9.ICU

As mentioned above, InterSystems Caché uses the ICU engine. Comprehensive documentation is available online:

- http://userguide.icu-project.org/strings/regexp
- http://userguide.icu-project.org/strings/regexp#TOC-Regular-Expression-Metacharacters
- http://userguide.icu-project.org/strings/regexp#TOC-Regular-Expression-Operators
- http://userguide.icu-project.org/strings/regexp#TOC-Replacement-Text
- http://userguide.icu-project.org/strings/regexp#TOC-Flag-Options

## 5.10.Tools

There are many tools out there that support developers in creating regular expressions – some of them free, others come with a commercial license. My personal choice is RegexBuddy (http://www.regexbuddy.com/) – it provides a comprehensive set of interactive and visual features to create and test regular expressions in different flavors.



```
"[^"\\]*(?:\\.[^"\\]*)*"
```

History:
- Programming: String
- Programming: String (escape quotes)
- Programming: String (multiline; escape quotes)

- Java 8 & Java 4
- All selected applications handle your regular expression in the same way
- Match the character """ literally
- Match any single character NOT present in the list below
  - Between zero and unlimited times, as many times as possible, giving ba
  - The literal character """
  - The backslash character
- Match the regular expression below
  - Between zero and unlimited times, as many times as possible, giving ba
  - Match the backslash character
  - Match any single character that is NOT a line break character (line feed
  - Match any single character NOT present in the list below
    - Between zero and unlimited times, as many times as possible, giving
    - The literal character """
    - The backslash character
- Match the character """ literally

```
"string"
"two""strings"
"multi
 line
 string"
"string with \"quotes\""
"multi
 line
 string
 with
 \"quotes\""
```

```
                        Start Length
Match 1 of 6:  "string"     0      8
```

#Best Practices #ObjectScript #Tutorial #Caché #InterSystems IRIS

**Source URL:**https://community.intersystems.com/post/using-regular-expressions-cach%C3%A9