DZone Refcardz

CONTENTS INCLUDE:

- About Regular Expressions
- Advanced Features
- The Dot Operator
- Quantifiers
- Flags
- Character Classes... and more!

Regular Expressions

By: Callum Macrae

ABOUT REGULAR EXPRESSIONS

A regular expression, also known as a regex or regexp, is a way of defining a search pattern. Think of regexes as wildcards on steroids. Using wildcards, *.txt matches anything followed by the string ".txt". But regular expressions let you e.g.: match specific characters, refer back to previous matches in the expression, include conditionals within the expression, and much more.

This Refcard assumes basic familiarity with program structures, but no prior knowledge of regular expressions. It would be wise to play with as many of the examples as possible, since mastering regex requires a good deal of practical experience.

There is no single standard that defines what regular expressions must and must not do, so this Refcard does not cover every regex version or 'flavor'. Parts of this Refcard will apply to some flavors (for example, PHP and Perl), while the same syntax will throw parse errors with other flavors (for example, JavaScript, which has a surprisingly poor regex engine).

Regular expressions are supported in most programming and scripting languages, including Java, .NET, Perl, Python, PHP, JavaScript, MySQL, Ruby, and XML. Languages that do not natively support regular expressions nearly always have libraries available to add support. If you prefer one regex flavor to the flavor the language natively supports, libraries are often available which will add support for your preferred flavor (albeit often using a slightly different syntax to the native expressions).

The Structure of a Regular Expression

All regular expressions follow the same basic structure: expression plus flag. In the following example, "regexp?" is the expression, and "mg" is the flag used.

```
/regexp?/mg
```

Certain programming languages may also allow you to specify your expression as a string. This allows you to manipulate the expression programmatically – something you are very likely to do, once you get the hang of writing regexes. For example, the following code will specify the same regular expression as above in JavaScript:

```
new RegExp("regexp?", "mg"); // returns /regexp?/mg
```

BASIC FEATURES

Normal Characters

A regular expression containing no special characters ($()*+.?[^|) will match exactly what is contained within the expression. The following expression will match the "world" in "Hello world!":\

```
"Hello world!".match(/world/); // "world"
```

These characters are treated as special characters:

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

and need to be escaped using the backslash ("\") character:

```
"Price: $20".match(/\$20/); // "$20"
```

(For more on handling special characters, see page 2 below.)

Character Types

Regex includes character types that can be used to match a range of commonly used characters:

Type	Description
\d	Decimal digit: [0-9]
\D	Not a decimal digit: [^0-9]
\s	Whitespace character: [\n\r\t\f]
\S	Not a whitespace character: [^ \n\r\t\f]
\w	Word character: [a-zA-Z0-9_]
\W	Not a word character: [^a-zA-Z0-9_]

Comments

Comments in regex are inserted like this:

```
"Hello world!".match(/wor(?#comment)ld/); // "world"
```

Many flavors of regex (including Java, .Net, Perl, Python, Ruby, and XPath) include a free-spacing mode (enabled using the x flag). This lets you use just a hash character to start a comment:

```
/
        (0?\d|1[0-2])       # Month
        \/                  # Separator
        (20\d\d)  # Year between 2000 and 2099
/x
```

To provide immediate feedback on regex match results, this Refcard will include matches in comments, like this:

```
"Regex is awesome".match(/regexp?/i); // "Regex"
```

The Dot Operator

The dot character (".") can be used to match any character except for new line characters (\n and \r). Be careful with the dot operator: it may match more than you would intuitively guess.

For example, the following regular expression will match the letter c, followed by any character that isn't a new line, followed by the letter t:

```
"cat".match(/c.t/); // "cat"
```

The dot operator will only match one character ("a").

Contrast this simple usage of the dot operator with the following, dangerous usage:

```
/<p class="(.+)">/
```

That expression initially looks like it would match an opening HTML paragraph tag, make sure that the only attribute is the class attribute, and store the class attribute in a group. The "+" is used to quantify the dot operator: the "+" says 'match one or more'. Most of the expression is treated as literal text, and the (.+) looks for a string of any character of any length and stores it in a group. However, it doesn't work. Here's what it matches instead:

```
"<p class='test' onclick='evil()'>".match(/<p class='(.+)'>/);
          // ["<p class='test' onclick='evil()'>",
          //        "test' onclick='evil()"]
```

Instead of matching just the class, it has matched the on-click attribute. This is because the dot operator, matched with the "+" quantifier, picked up everything before the closing greater-than. The first lesson: the dot operator can be a clumsy weapon. Always use the most targeted feature of regex you can. The second lesson: quantifiers are ex-tremely powerful, especially with the dot operator.

The Pipe Character

The pipe character causes a regular expression to match either the left side or the right side. For example, the following expression will match either "abc" or "def":

```
/abc|def/
```

Anchors

Regex anchors match specific, well-defined points, like the start of the expression or the end of a word. Here are the most important:

Char	Description	Example
^	The caret character matches the beginning of a string or the beginning of a line in multi-line mode.	/^./ Matches "a" in "abc\ndef", or both "a" and "d" in mul-ti-line mode.
$	The dollar character matches the end of a string, or the end of a line in multi-line mode.	/.$/ Matches "f" in "abc\ndef", or both "c" and "f" in mul-ti-line mode.
\A	Always matches the start of a string, irrespective of multi-line mode.	/\A./ Matches "a" in "abc\ndef".
\Z	Always matches the end of a string, irrespective of multi-line mode. If the last character is a line break, it will match the character before.	/.\Z/ Matches "f" in "abc\ ndef\n". Matches "f" in "abc\ndef"
\z	Always matches the end of a string, irrespective of multi-line mode. Will always match the last character.	/.\z/ Matches "f" in "abc\ndef". Matches "\n" in "abc\ ndef\n"
\b	Matches a word boundary: the position between a word character (\w) and a non-word character or the start or end of the string (\ W|^|$).	/.\b/ Matches "o", " " and "d" in "Hello world".
\B	Matches a non-word boundary: the position between two word characters or two non-word characters.	/\B.\B/ Matches every letter but "H" and "o" in "Hello".
\<	Matches the start of a word - similar to \b, but only at the start of the word.	/\<./ Matches "H" and "w" in "Hello world".
\>	Matches the end of a word - similar to \b, but only at the end of the word.	/.\>/ Matches "o" and "d" in "Hello world".

GROUPS

Regex groups lump just about anything together. Groups are commonly used to match part of a regular expression and refer back to it later, or to repeat a group multiple times.

In this Refcard, groups will be represented like this:

```
"Regex is awesome".match(/Regex is (awesome)/);
          // ["Regex is awesome", "awesome"]
```

Capturing Groups

The most basic group, the capturing group, is marked simply by putting parentheses around whatever you want to group:

```
"Hello world!".match(/Hello (\w+)/); // ["Hello world", "world"]
```

(Note the \w character type, introduced on page 1.) Here the capturing group is saved and can be referred to later.

Non-capturing Groups

There are also non-capturing groups, which will not save the matching text and so cannot be referred back to later. We can get these by putting "?:" after the opening parenthesis of a group:

```
"Hello world!".match(/Hello (?:\w+)/); // ["Hello world"]
```

Using Capturing vs. Non-capturing Groups

Use capturing groups only when you will need to refer to the grouped text later. Non-capturing groups save memory and simplify debugging by not filling up the returned array with data that you don't need.

Named Capture Groups

When a regex includes a named capture group, the func-tion will return an object such that you can refer to the matched text using the name of the group as the property:

```
"Hello world!".match(/Hello (?<item>\w+)/);
          // {"0": "Hello world", "1": "world", "item": "world"}
```

The name of the group is surrounded by **angle brackets.**

Backreferences

You can refer back to a previous capturing group by using a backreference. After matching a group, refer back to the text matched in that group by inserting a backward slash followed by the number of the capturing group. For example, the following will match the same character four times in a row:

```
"aabbbb".match(/(\w)\1{3}/); // bbbb
```

In this example, the group matches the first "b"; then \1 refers to that "b" and matches the letters after it three times (the "3" inside the curly brackets). Capturing groups are numbered in numerical order starting from 1.

A more complicated (but quite common) use-case involves matching repeated words. To do this, we first want to match a word. We'll match a word by grabbing a word boundary, followed by a group capturing the first word, followed by a space, followed by a backreference to match the second word, and then a word boundary to make sure that we're not matching word-parts plus whole words:

```
"hello hello world!".match(/\b(\w+) \1\b/); "hello hello"
```

It is possible to have a backreference refer to a named capture group, too, by using \k<groupname>. The following regular expression will do the same as above:

```
/\b(?<word>\w+) \k<word>\b/
```

Groups and Operator Precedence

Groups can be used to control effective order of operations. For example, the pipe character has the lowest operator precedence, so pipes are often used within groups to con-trol the insertion point of the logical 'or'. The following expression will match either "abcdef" or "abcghi":

```
/abc(def|ghi)/
```

CHARACTER CLASSES

A character class allows you to match a range or set of characters. The following uses a character class to match any of the contained letters (in this case, any vowel):

```
/[aeiou]/
```

The following regex will match "c", followed by a vowel, followed by "t"; it will match "cat" and "cot", but not "ctt" or "cyt":

```
/c[aeiou]t/
```

You can also match a range of characters using a character class. For example, /[a-i]/ will match any of the letters between a and i (inclusive).

Character classes work with numbers too. This is particularly useful for matching e.g. date-ranges. The following regular expression will match a date between 1900 and 2013:

```
/19[0-9][0-9]|200[0-9]|201[0-3]/
```

Ranges and specific matches can be combined: /[a-mxyz]/ will match the letters a through m, plus the letters x, y, and z.

Negated Character Classes

We can also use character classes to specify characters we don't want to match. These are called negated character classes, and are created by putting a caret ("^") at the be-ginning of the class. This indicates that we want to match a character that doesn't match any of the characters in a character class.

The following negated character class will match a "c", followed by a non-vowel, followed by a "t". This means that it will match "cyt", "c0t" and "c.t", but will not match "cat", "cot", "ct" or "cyyt":

```
/c[^aeiou]t/
```

Hot Tip Be careful to distinguish between negated character classes ("match not-x") and simply not matching a character ("do not match x").

QUANTIFIERS

Quantifiers are used to repeat something multiple times. The basic quantifiers are: + * ?

The most basic quantifier is "+", which means that something should be repeated one or more times. For example, the following expression matches one or more vowels:

```
"queueing".match(/[aeiou]+/); // "ueuei"
```

Note that it doesn't match the previous match (so not the first "u" multiple times), but rather the previous expression, i.e. another vowel. In order to match the previous match, use a group and a backreference, along with the "*" operator.

The "*" operator is similar to the "+" operator, except that "*" matches zero or more instead of one or more of an expression.

```
"ct".match(/ca*t/); // "ct"
"cat".match(/ca*t/); // "cat"
"caat".match(/ca*t/); // "caat"
```

To match the same character from a match one or more times, use a group, a backreference, and the "*" operator:

```
"saae".match(/([aeiou])\1*/); // "aa"
```

In the string "queuing," this expression would match only "u". In "aaeaa", it would match only the first "aa".

The "?" character is the final basic quantifier. It makes the previous expression optional:

```
"ct".match(/ca?t/); // "ct"
"cat".match(/ca?t/); // "cat"
"caat".match(/ca?t/); // no match
```

We can make more powerful quantifiers using curly braces. The following expressions will both match exactly five oc-currences of the letter "a".

```
/^a{5}$/
/^aaaaa$/
```

The first is a lot more readable, especially with higher numbers or longer groups, where it would be impractical to type them all out.

We can use curly braces to repeat something between a range of times:

```
/^a{3,5}$/
```

That will match the letter "a" repeated 3, 4, or 5 times.

If you want to match something repeated up to a certain number of times, you can use 0 as the first number. If you want to match something more than a certain number with no maximum, you can just leave the second number blank:

```
/^a{3,}$/
```

This will match the letter "a" repeated 3 or more times with no maximum number of repetitions.

Greedy vs. Lazy Matches

There are two types of matches: greedy or lazy. Most flavors of regular expressions are greedy by default.

To make a quantifier lazy, append a question mark to it.

The difference is best illustrated using an example:

```
'He said "Hello" and then "world!"'.match(/".+"/);
          // "Hello" and then "world!"
'He said "Hello" and then "world!"'.match(/".+?"/);
          // "Hello"
```

The first match is greedy. It has matched as many characters as it can (between the first and last quotes). The second match is lazy. It has matched as few characters as possible (between the first and second quotes).

Note that the question mark does not make the quantifier optional (which is what the question mark is usually used for in regular expressions).

More examples of greedy vs. lazy quantification:

```
'He said "Hello" and then "world!"'.match(/".+?"/);
          // "Hello"
'He said "Hello" and then "world!"'.match(/".*?"/);
          // "Hello"

"aaa".match(/aa?a/); // "aaa"
"aaa".match(/aa??a/); // "aa"

'He said "Hello" and then "world!"'. match(/".{2,}?"/);
          // "Hello"
```

Note the difference the question mark makes.

Hot Tip Be aware of differences between greedy and lazy matching when writing regular expressions. A greedy expression can be rewritten into a shorter lazy expression. For regex beginners, the difference often accounts for unexpected, counterintuitive results.

Flags

Flags (also known as "modifiers") can be used to modify how the regular expression engine will treat the regular expression. In most flavors of regular expression, flags are specified after the final forward slash, like this:

```
/regexp?/mg
```

In this case, the flags are m and g.

Flag support does vary among regex flavors, so check the documentation for the regex engine in your language or library.

The most common flags are: i, g, m, s, and x.

The i flag is the case insensitive flag. This tells the regex engine to ignore the case. For example, the character class [a-z] will match [a-zA-Z] when the i flag is used and the literal character "c" will also match C.

```
"CAT".match(/c[aeiou]t/); // no match
"CAT".match(/c[aeiou]t/i); // "CAT"
```

The g flag is the global flag. This tells the regex engine to match every instance of the match, not just the first (which is the default). For example, when replacing a match with a string in JavaScript, using the global flag means that every match will be replaced, not just the first. The following code (in JavaScript) demonstrates this:

```
"cat cat".replace(/cat/, "dog"); // "dog cat"
"cat cat".replace(/cat/g, "dog"); // "dog dog"
```

The m flag is the multiline flag. This tells the regex engine to match "^" and "$" to the beginning and end of a line respectively, instead of the beginning and end of the string (which is the default). Note that \A and \Z will still only match the beginning and end of the string, regardless of what multi-line mode is set to.

In the following code example, we see a lazy match from ^ to $ (we're using [\s\S] because the dot operator doesn't match new lines by default). When ^ and $ match the beginning of the line (when multi-line mode is enabled), it returns only one line of text. Otherwise, it returns everything.

```
"1st line\n2nd line".match(/^[\s\S]+?$/); // "1st line\n2nd line"
"1st line\n2nd line".match(/^[\s\S]+?$/m); // "1st line"
```

The s flag is the singleline flag (also called the dotall flag in some flavors). By default, the dot operator will match every character except for new lines. With the s flag enabled, the dot operator will match everything, including new lines.

```
"This is\na test".match(/^.+$/); // no match
"This is\na test".match(/^.+$/s); // "This is\na test"
```

Hot Tip This behavior can be simulated in languages where the s flag isn't supported by using [\s\S] instead of the s flag.

The x flag is the extended mode flag, which activates extended mode. Extended mode allows you to make your regular expressions more readable by adding whitespace and comments. Consider the following two expressions, which both do exactly the same thing, but are drastically different in terms of readability.

```
/([01]?\d{1,3}|200\d|201[0-3])([-/.])(0?[1-9]|1[012])\2(0?[1-9]|[12]\d|3[01])/

/
          ([01]?\d{1,3}|200\d|201[0-3]) # year
          ([-/.])                       # separator
          (0?[1-9]|1[012])              # month
          \2                            # same separator
          (0?[1-9]|[12]\d|3[01])        # date
/x
```

Modifying Flags Mid-Expression

In some flavors of regular expressions, it is possible to modify a flag mid-expression using a syntax similar (but not identical) to groups.

To enable a flag mid-expression, use "(?i)" where i is the flag you want to activate:

```
"foo bar".match(/foo bar/); // "foo bar"
"foo BAR".match(/foo bar/); // no match
"foo bar".match(/foo (?i)bar/); // "foo bar"
"foo BAR".match(/foo (?i)bar/); // "foo BAR"
```

To disable a flag, use "(?-i)", where i is the flag you want to disable:

```
"foo bar".match(/foo bar/i); // "foo bar"
"foo BAR".match(/foo bar/i); // "foo BAR"
"foo bar".match(/foo (?-i)bar/i); // "foo bar"
"foo BAR".match(/foo (?-i)bar/i); // no match
```

You can activate a flag and then deactivate it:

```
/foo (?i)bar(?-i) world/
```

And you can activate or deactivate multiple flags at the same time by using "(?ix-gm)" where i and x are the flags we want enabling and g and m the ones we want disabling.

HANDLING SPECIAL CHARACTERS

To handle special characters in a regular expression, either (a) use their ASCII or Unicode numbers (in a variety of forms), or (b) just escape them.

Any character that isn't $()*+.?[\]^{|} will match itself, even if it is a special character. For example, the following will match an umlaut:

```
/ß/
```

To match any of the characters mentioned above ()$()*+.?[\]^{|}() you can prepend them with a backward slash to mark them as literal characters:

```
/\$/
```

It is impractical to put some special characters, such as emoji (which will not display in most operating systems) and invisible characters, right into a regular expression. To solve this problem, when writing regular expressions just use their ASCII or Unicode character codes.

The first ASCII syntax allows you to specify the ASCII character code as octal, which must begin with a 0 or it will be parsed as a backreference. To match a space, we can use this:

```
/\040/
```

You can match ASCII characters using hexadecimal, too. The following will also match a space:

```
/\x20/
```

You can also match characters using their Unicode character codes, but only as hexadecimal. The following will match a space:

```
/\u0020/
```

A few characters can also be matched using shorter escapes. We saw a few of them previously (such as \t and \n for tab and new line characters, respectively – pretty standard), and the following table defines all of them properly:

Escape	Description	Character code
\a	The alarm character (bell).	\u0007
\b	Backspace character, but only when in a character class-word boundary otherwise.	\u0008
\e	Escape character.	\u001B
\f	Form feed character.	\u000C
\n	New line character.	\u000A
\r	Carriage return character.	\u000D
\t	Tab character.	\u0009
\v	Vertical tab character.	\u000B

Finally, it is possible to specify control characters by pre-pending the letter of the character with \c. The following will match the character created by ctrl+c:

```
/\cC/
```

ADVANCED FEATURES

Advanced features vary more widely by flavor, but several are especially useful and worth covering here.

POSIX Character Classes

POSIX character classes are a special type of character class which allow you to match a common range of characters—such as printing characters—without having to type out an ordinary character class every time.

The table below lists all the POSIX character classes available, plus their normal character class equivalents.

Class	Description	Equivalent
[:alnum:]	Letters and digits	[a-zA-Z0-9]
[:alpha:]	Letters	[a-zA-Z]
[:ascii:]	Character codes 0 - x7F	[\x00-\x7F]
[:blank:]	Space or tab character	[\t]
[:cntrl:]	Control characters	[\x00-\x1F\x7F]
[:digit:]	Decimal digits	[0-9]
[:graph:]	Printing characters (minus spaces)	[\x21-\x7E]
[:lower:]	Lower case letters	[a-z]
[:print:]	Printing characters (including spaces)	[\x20-\x7E]
[:punct:]	Punctuation: [:graph:] minus [:alnum:]	
[:space:]	White space	[\t\r\n\v\f]
[:upper:]	Upper case letters	[A-Z]
[:word:]	Digits, letters, and underscores	[a-zA-Z0-9_]
[:xdigit:]	Hexadecimal digits	[a-fA-F0-9]

ASSERTIONS

Lookahead Assertions

Assertions match characters but don't return them. Consider the expression "\>", which will match only if there is an end of word (i.e., the engine will look for (\W|$)), but won't actually return the space or punctuation mark following the word.

Assertions can accomplish the same thing, but explicitly and therefore with more control.

For a positive lookahead assertion, use "(?=x)".

The following assertion will behave exactly like \>:

```
/.(?=\W|$)/
```

This will match the "o" and "d" in the string "Hello world" but it will not match the space or the end.

Consider the following, more string-centered example. This expression matches "bc" in the string "abcde":

```
abcde".match(/[a-z]{2}(?=d)/); // "bc"
```

Or consider the following example, which uses a normal capturing group:

```
"abcde".match(/[a-z]{2}(d)/); // ["bcd", "d"]
```

While the expression with the capturing group matches the "d" and then returns it, the assertion matches the character and does not return it. This can be useful for backrefer-ences or to avoid cluttering up the return value.

The assertion is the "(?=d)" bit. Pretty much anything you would put in a group in a regular expression can be put into an assertion.

Negative Lookahead Assertions

Use a negative lookahead assertion ("(?!x)") when you want to match something that is not followed by something else. For example, the following will match any letter that is not followed by a vowel:

This expression returns e, l, and o in hello. Here we notice a difference between positive assertions and negative assertions. The following positive lookahead does not behave the same as the previous negative lookahead:

```
/\w(?=[^aeiou])/
```

That would match only the e and the l in hello. A negative assertion does not behave like a negated character class. A negative assertion simply means that the expression contained within the negative assertion should not be matched—not that something that doesn't match the nega-tive assertion should be present.

Lookbehind Assertions

As with lookahead assertions, you can use negative look-behind assertions to specify that something should not be there. There are two syntaxes for doing this; the following two expressions both look for a letter that does not have a vowel before it.

```
/(?<![aeiou]+)\w/
/(?!=[aeiou]+)\w/
```

As with negative lookahead assertions, this expression looks for letters that do not have vowels before them – not letters with non-vowels before them. You can have multiple assertions in a row, for example to assert three digits in a row that are not "123".

When you shouldn't use Regular Expressions

Regexes are amazing. Beginners are sometimes tempted to regexify everything. This is a mistake. In many cases, it is better to use a parser instead. Here are some such cases:

Parsing XML / HTML. There are many technical reasons why HTML and XML should never be parsed using regex, but all far too long winded for a cheatsheet - see the Chomsky hierarchy for an explanation. Instead of using regular expressions, you can simply use an XML or HTML parser instead, of which there are many available for almost every language.

Email addresses. The RFCs specifying what makes a valid email address and what does not are extremely complicated. A fairly famous regular expression which satisfies RFC-822 is over 6,500 characters long: (see http://www.ex-parrot.com/pdw/Mail-RFC822-Address.html). Instead of using regular expressions:

1. split the email address into two parts (address and domain)
2. check for control characters
3. check for domain validity

URLs. While there are standards (e.g. http://url.spec.whatwg.org/), they are often broken (e.g. Wikipedia uses brackets in their URLs).

For string operations. If you're just checking whether a string contains a certain word, regular expressions are overkill. Instead, use a function designed specifically for working with strings.

RESOURCES

Huge library of community-contributed regexes, rated by users:
http://regexlib.com/

Complete tutorial for beginners, with exercises:
http://regex.learncodethehardway.org/book/

Regex Tuesday: over a dozen regex challenges, with varying levels of difficulty: http://callumacrae.github.io/regex-tuesday/