JavaScript
RegExp

/an example based guide/

RegExp: /\d+(?!foo|bar|baz)/i

Not followed by:

foo

bar

baz

Digit

1 or more times
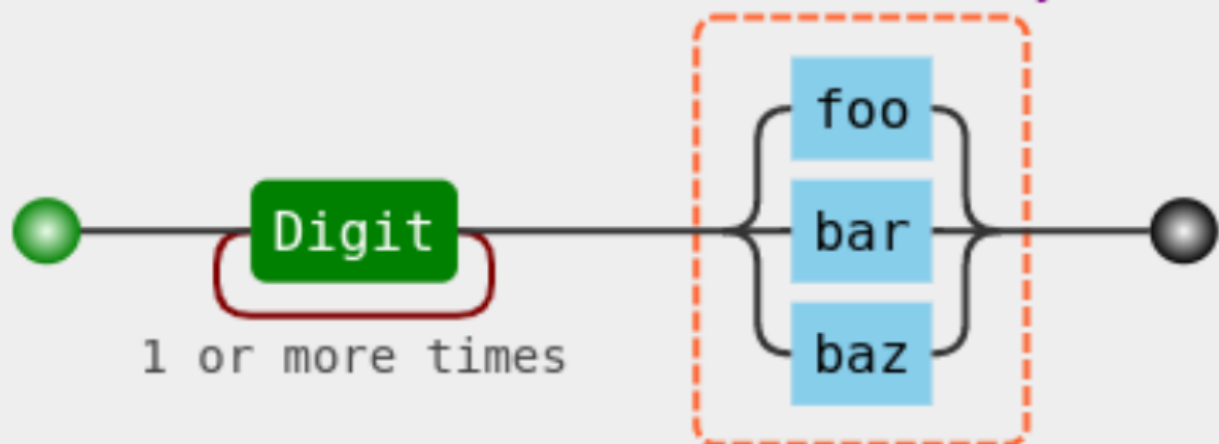
Sundeep Agarwal

Table of contents

Preface

Scripting and automation tasks often need to extract particular portions of text from input data or modify them from one format to another. This book will help you learn Regular Expressions as implemented in JavaScript. Regular expressions can be considered as a mini-programming language in itself and is well suited for a variety of text processing needs.

The book heavily leans on examples to present features of regular expressions one by one. It is recommended that you manually type each example and experiment with them. Understanding both the nature of sample input string and the output produced is essential. As an analogy, consider learning to drive a bike or a car — no matter how much you read about them or listen to explanations, you need to practice a lot and infer your own conclusions. Should you feel that copy-paste is ideal for you, code snippets are available chapter wise on GitHub.

Prerequisites

A good understanding of basic-level programming concepts and prior experience working with JavaScript. Should also know functional programming concepts like `map` and `filter`.

Conventions

- The examples presented here have been tested on Chrome/Chromium console (version 78+) and includes features not available in other browsers and platforms
- Code snippets shown are copy pasted from console and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines have been added to improve readability and output is skipped when it is `undefined` or otherwise unnecessary to be shown
- Unless otherwise noted, all examples and explanations are meant for *ASCII* characters
- External links are provided for further reading throughout the book. Not necessary to immediately visit them. They have been chosen with care and would help, especially during rereads
- The learn_js_regexp repo has all the code snippets and exercises and other details related to the book. Click the **Clone or download** button to get the files

Acknowledgements

- MDN: Regular Expressions — documentation and examples
- stackoverflow — for getting answers to pertinent questions on JavaScript and regular expressions
- tex.stackexchange — for help on `pandoc` and `tex` related questions
- Cover image: LibreOffice Draw and regulex
- Warning and Info icons by Amada44 under public domain

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/learn_js_regexp/issues

Goodreads: https://www.goodreads.com/book/show/49090622-javascript-regexp

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a freelance trainer, author and mentor. His previous experience includes working as a Design Engineer at Analog Devices for more than 5 years. You can find his other works, primarily focused on Linux command line, text processing, scripting languages and curated lists, at https://github.com/learnbyexample. He has also been a technical reviewer for Command Line Fundamentals book and video course published by Packt.

List of books: https://learnbyexample.github.io/books/

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

Code snippets are available under MIT License

Resources mentioned in Acknowledgements section above are available under original licenses.

Book version

1.0

See Version_changes.md to track changes across book versions.

What's so special about Regular Expressions?

Regular Expressions is a versatile tool for text processing. You'll find them included in standard library of most programming languages that are used for scripting purposes. If not, you can usually find a third-party library. Syntax and features of regular expressions vary from language to language. JavaScript's syntax is similar to that of Perl language, but there are significant feature differences.

The String object in JavaScript supports variety of methods to deal with text. So, what's so special about regular expressions and why would you need it? For learning and understanding purposes, one can view regular expressions as a mini programming language in itself, specialized for text processing. Parts of a regular expression can be saved for future use, analogous to variables. There are ways to perform AND, OR, NOT conditionals. Operations similar to range and repetition and so on.

Here's some common use cases:

- Sanitizing a string to ensure that it satisfies a known set of rules. For example, to check if a given string matches password rules.
- Filtering or extracting portions on an abstract level like alphabets, numbers, punctuation and so on.
- Qualified string replacement. For example, at the start or the end of a string, only whole words, based on surrounding text, etc.

Further Reading

- The true power of regular expressions — it also includes a nice explanation of what *regular* means in this context
- softwareengineering: Is it a must for every programmer to learn regular expressions?
- softwareengineering: When you should NOT use Regular Expressions?
- codinghorror: Now You Have Two Problems
- wikipedia: Regular expression — this article includes discussion on regular expressions as a formal language as well as details on various implementations

RegExp introduction

This chapter will get you started with defining RegExp objects and using them inside string methods. To keep it simple, the examples will not use special characters related to regular expressions. The main focus will be to get you comfortable with syntax and text processing examples. Two methods will be introduced in this chapter. The `test` method to search if the input contains a string and the `replace` method to substitute a portion of the input with something else.

> ℹ This book will use **regular expressions** and **regexp** interchangeably. When specifically referring to a JavaScript object, **RegExp** will be used.

Console

As mentioned in the preface, examples presented in this book have been tested on Chrome/Chromium console. Other browsers based on Chromium may also work. Use `Ctrl+Shift+J` shortcut from a new tab to open a console. Some variable names are same across different chapters, so open another tab in such cases. See MDN: Regular Expressions for documentation, examples and feature compatibility details.

test method

First up, a simple example to test whether a string is part of another string or not. Normally, you'd use the `includes` method and pass a string as argument. For regular expressions, use the `test` method on a RegExp object, which is defined by the search string enclosed within `//`.

```
> let sentence = 'This is a sample string'

// check if 'sentence' contains the given string argument
> sentence.includes('is')
< true
> sentence.includes('z')
< false

// check if 'sentence' matches the pattern as described by the regexp
> /is/.test(sentence)
< true
> /z/.test(sentence)
< false
```

Here's some examples of using the `test` method in conditional expressions.

```
> let report = 'string theory'

> if (/ring/.test(report)) {
      console.log('mission success')
  }
< mission success
```

```
> if (!/fire/.test(report)) {
      console.log('mission failed')
  }
< mission failed
```

And here's some array processing examples.

```
> let words = ['cat', 'attempt', 'tattle']

// get all elements that contain 'tt'
> words.filter(w => /tt/.test(w))
< ["attempt", "tattle"]

// check if all the elements contain 'at'
> words.every(w => /at/.test(w))
< true

// check if any element contains 'stat'
> words.some(w => /stat/.test(w))
< false
```

Flags

Some of the regular expression functionality is enabled by passing flags, represented by an alphabet character. If you have used command line, flags are similar to command options, for example `grep -i` will perform case insensitive matching.

In this chapter, two flags will be discussed:

- `i` flag to ignore case while matching alphabets (default is case sensitive matching)
- `g` flag to match all occurrences (by default only the first one is matched)

Examples for `i` flag is shown below. `g` flag will be discussed in replace method section later in this chapter.

```
> /cat/.test('CaT')
< false
> /cat/i.test('CaT')
< true

> ['Cat', 'cot', 'CATER', 'SCat', 'ScUtTLe'].filter(w => /cat/i.test(w))
< ["Cat", "CATER", "SCat"]
```

RegExp constructor and reuse

The RegExp object can be saved in a variable. This helps to improve code clarity, enable reuse, etc.

```
> const pet = /dog/
```

```
> pet.test('They bought a dog')
< true
> pet.test('A cat crossed their path')
< false
```

RegExp objects can also be constructed using the `RegExp()` constructor. The first argument is either a string or a RegExp object. The second argument is used to specify one or more flags.

```
> const pat = new RegExp('dog')
> pat
< /dog/

// if flags are needed, specify them as the second argument
> new RegExp('dog', 'i')
< /dog/i
```

The main advantage of the constructor over `//` literal is the ability to dynamically construct the regexp using `${}` to insert content of other variables or the result of an expression.

```
> let greeting = 'hi'

> const pat1 = new RegExp(`${greeting} there`)
> pat1
< /hi there/

> new RegExp(`${greeting.toUpperCase()} there`)
< /HI there/
```

⚠️ Both strings and RegExp objects support special escape sequences. Some sequences like `\t` and `\n` mean the same but sequences like `\b` have completely different meaning. To avoid unexpected behavior, use `\\` instead of `\`.

```
> new RegExp('123\tabc')
< /123   abc/

> new RegExp('123\\tabc')
< /123\tabc/
```

Here's a console screenshot of another example.

replace method

The `replace` string method is used for search and replace operations.

```
// change only first match
> '1,2,3,4'.replace(/,/, '-')
< "1-2,3,4"

// change all matches by adding 'g' flag
> '1,2,3,4'.replace(/,/g, '-')
< "1-2-3-4"

// multiple flags can be combined
> 'cArT PART tart mArt'.replace(/art/ig, '2')
< "c2 P2 t2 m2"
```
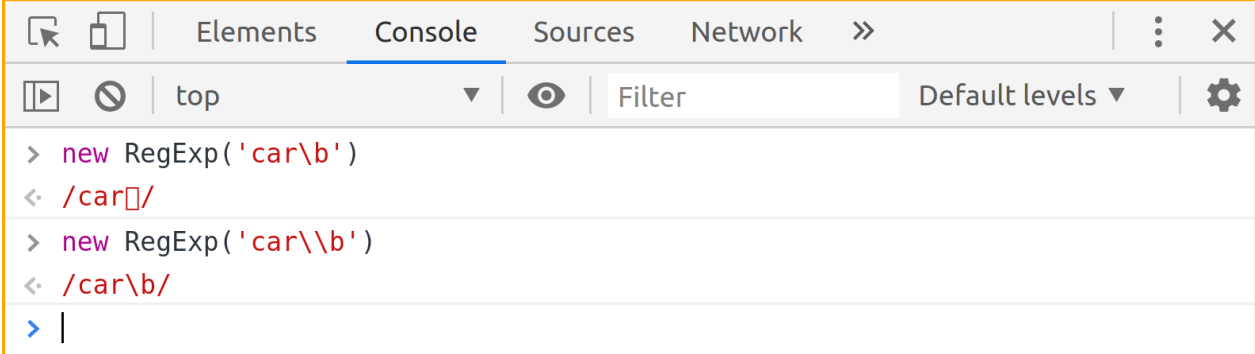
⚠️ A common mistake is forgetting that strings are immutable. If you want to save changes to the same variable, you need to explicitly assign the result back to that variable.

```
> let word = 'cater'
// this will return a string but won't modify the 'word' variable
> word.replace(/cat/, 'hack')
< "hacker"
> word
< "cater"

// need to explicitly assign the result for in-place modification
> word = word.replace(/cat/, 'hack')
< "hacker"
> word
< "hacker"
```

Cheatsheet and Summary

Note	Description
MDN: Regular Expressions	MDN documentation for JavaScript regular expressions
`/pat/`	a RegExp object
`const p1 = /pat/`	save regexp in a variable for reuse, clarity, etc
`/pat/.test(s)`	Check if given pattern is present anywhere in input string
	returns `true` or `false`
`i`	flag to ignore case when matching alphabets
`g`	flag to match all occurrences
`new RegExp('pat', 'i')`	construct RegExp from a string
	second argument specifies flags
	use backtick strings with `${}` for interpolation
`s.replace(/pat/, 'repl')`	method for search and replace

This chapter introduced how to define RegExp objects and use them with `test` and `replace` methods. You also learnt how to use flags to change the default behavior of regexps. The examples presented were more focused on introducing text processing concepts. Next chapter onwards, you'll learn regular expression syntax and features.

Exercises

ⓘ All the exercises are also collated together in one place at Exercises.md.

a) Check if the given input strings contain `two` irrespective of case.

```
> let s1 = 'Their artwork is exceptional'
> let s2 = 'one plus tw0 is not three'
> let s3 = 'TRUSTWORTHY'

> const p1 =        // add your solution here

> p1.test(s1)
< true
> p1.test(s2)
< false
> p1.test(s3)
< true
```

b) For the given multiline input string, filter all lines NOT containing the string `2`.

```
> let purchases = `items qty
apple 24
mango 50
guava 42
onion 31
water 10`

> const num =       // add your solution here

> console.log(purchases.split('\n')
                       .filter()        // add your solution here
                       .join('\n'))
< items qty
  mango 50
  onion 31
  water 10
```

ⓘ You'd be able to solve this using just `replace` method by the end of Dot metacharacter and Quantifiers chapter.

c) Replace all occurrences of `0xA0` with `0x50` for the given input string.

```
> let h1 = 'start address: 0xA0, end address: 0xA0B'
> h1.replace()       // add your solution here
< "start address: 0x50, end address: 0x50B"
```

Anchors

In this chapter, you'll be learning about qualifying a pattern. Instead of matching anywhere in the given input string, restrictions can be specified. For now, you'll see the ones that are already part of regular expression features. In later chapters, you'll learn how to define your own rules for restriction.

These restrictions are made possible by assigning special meaning to certain characters and escape sequences. The characters with special meaning are known as **metacharacters** in regexp parlance. In case you need to match those characters literally, you need to escape them with a `\` (discussed in Escaping metacharacters chapter).

String anchors

This restriction is about qualifying a regexp to match only at start or end of an input string. These provide functionality similar to the string methods `startsWith` and `endsWith`. First up is `^` metacharacter, which restricts the matching to the start of string.

```
// ^ is placed as a prefix to the pattern
> /^cat/.test('cater')
< true
> /^cat/.test('concatenation')
< false

> /^hi/.test('hi hello\ntop spot')
< true
> /^top/.test('hi hello\ntop spot')
< false
```

To restrict the match to the end of string, `$` metacharacter is used.

```
// $ is placed as a suffix to the pattern
> /are$/.test('spare')
< true
> /are$/.test('nearest')
< false

> let words = ['surrender', 'unicorn', 'newer', 'door', 'empty', 'eel', 'pest']
> words.filter(w => /er$/.test(w))
< ["surrender", "newer"]
> words.filter(w => /t$/.test(w))
< ["pest"]
```

Combining both the start and end string anchors, you can restrict the matching to the whole string. Similar to comparing strings using the `==` operator.

```
> /^cat$/.test('cat')
< true
> /^cat$/.test('cater')
< false
```

The anchors can be used by themselves as a pattern. Helps to insert text at the start or end of string, emulating string concatenation operations. These might not feel like useful capability, but combined with other regexp features they become quite a handy tool.

```
> 'live'.replace(/^/, 're')
< "relive"
> 'send'.replace(/^/, 're')
< "resend"

> 'cat'.replace(/$/, 'er')
< "cater"
> 'hack'.replace(/$/, 'er')
< "hacker"
```

Line anchors

A string input may contain single or multiple lines. The newline character `\n` is used as the line separator. When the `m` flag is used, the `^` and `$` anchors will match the start and end of every line respectively.

```
// check if any line in the string starts with 'top'
> /^top/m.test('hi hello\ntop spot')
< true

// check if any line in the string ends with 'er'
> /er$/m.test('spare\npar\nera\ndare')
< false

// check if any complete line in the string is 'par'
> /^par$/m.test('spare\npar\nera\ndare')
< true
```

Just like string anchors, you can use the line anchors by themselves as a pattern.

```
> let items = 'catapults\nconcatenate\ncat'

> console.log(items.replace(/^/gm, '* '))
< * catapults
  * concatenate
  * cat

> console.log(items.replace(/$/gm, '.'))
< catapults.
  concatenate.
  cat.
```

> ⚠️ If there is a newline character at the end of string, there is an additional start/end of line match after the newline.

```
// 'foo ' is inserted three times
> '1\n2\n'.replace(/^/mg, 'foo ')
```

```
< "foo 1
  foo 2
  foo "

> '1\n2\n'.replace(/$/mg, ' baz')
< "1 baz
  2 baz
   baz"
```

⚠️ If you are dealing with Windows OS based text files, you'll have to convert `\r\n` line endings to `\n` first. Otherwise, you'll get end of line matches for both `\r` and `\n` characters. You can also handle this case in regexp by making `\r` as optional character with quantifiers (see Greedy quantifiers section).

Word anchors

The third type of restriction is word anchors. Alphabets (irrespective of case), digits and the underscore character qualify as word characters. You might wonder why there are digits and underscores as well, why not only alphabets? This comes from variable and function naming conventions — typically alphabets, digits and underscores are allowed. So, the definition is more oriented to programming languages than natural ones.

The escape sequence `\b` denotes a word boundary. This works for both the start of word and end of word anchoring. Start of word means either the character prior to the word is a non-word character or there is no character (start of string). Similarly, end of word means the character after the word is a non-word character or no character (end of string). This implies that you cannot have word boundary `\b` without a word character.

```
> let sample = 'par spar apparent spare part'

// replace 'par' irrespective of where it occurs
> sample.replace(/par/g, 'X')
< "X sX apXent sXe Xt"
// replace 'par' only at the start of word
> sample.replace(/\bpar/g, 'X')
< "X spar apparent spare Xt"
// replace 'par' only at the end of word
> sample.replace(/par\b/g, 'X')
< "X sX apparent spare part"
// replace 'par' only if it is not part of another word
> sample.replace(/\bpar\b/g, 'X')
< "X spar apparent spare part"
```

You can get lot more creative with using word boundary as a pattern by itself.

```
// sample = 'par spar apparent spare part'
// space separated words to double quoted csv
// note that 'replace' method is used twice here
> console.log(sample.replace(/\b/g, '"').replace(/ /g, ','))
< "par","spar","apparent","spare","part"
```

```
// make a programming statement more readable
// shown for illustration purpose only, won't work for all cases
> 'foo_baz=num1+35*42/num2'.replace(/\b/g, ' ')
< " foo_baz = num1 + 35 * 42 / num2 "
// excess space at start/end of string can be trimmed off
// later you'll learn how to add a qualifier so that trim is not needed
> 'foo_baz=num1+35*42/num2'.replace(/\b/g, ' ').trim()
< "foo_baz = num1 + 35 * 42 / num2"
```

The word boundary has an opposite anchor too. `\B` matches wherever `\b` doesn't match. This duality will be seen with some other escape sequences too. Negative logic is handy in many text processing situations. But use it with care as you might end up matching things you didn't intend!

```
// sample = 'par spar apparent spare part'

// replace 'par' if it is not start of word
> sample.replace(/\Bpar/g, 'X')
< "par sX apXent sXe part"
// replace 'par' at the end of word but not whole word 'par'
> sample.replace(/\Bpar\b/g, 'X')
< "par sX apparent spare part"
// replace 'par' if it is not end of word
> sample.replace(/par\B/g, 'X')
< "par spar apXent sXe Xt"
// replace 'par' if it is surrounded by word characters
> sample.replace(/\Bpar\B/g, 'X')
< "par spar apXent sXe part"
```

Here's some standalone pattern usage to compare and contrast the two word anchors.

```
> 'copper'.replace(/\b/g, ':')
< ":copper:"
> 'copper'.replace(/\B/g, ':')
< "c:o:p:p:e:r"

> '-----hello-----'.replace(/\b/g, ' ')
< "----- hello -----"
> '-----hello-----'.replace(/\B/g, ' ')
< " - - - - -h e l l o- - - - - "
```

Cheatsheet and Summary

Note	Description
metacharacter	characters with special meaning in regexp
^	restricts the match to start of string
$	restricts the match to end of string
\n	line separator, dos-style files may need special attention

Note	Description
	as they use `\r\n` as line endings
`m`	flag to match the start/end of line with `^` and `$` anchors
`\b`	restricts the match to start/end of words
	word characters: alphabets, digits, underscore
`\B`	matches wherever `\b` doesn't match

In this chapter, you've begun to see building blocks of regular expressions and how they can be used in interesting ways. But at the same time, regular expression is but another tool in the land of text processing. Often, you'd get simpler solution by combining regular expressions with normal string methods. Practice, experience and imagination would help you construct creative solutions. In coming chapters, you'll see more applications of anchors with other regexp features.

Exercises

a) Check if the given input strings contain `is` or `the` as whole words.

```
> let str1 = 'is; (this)'
> let str2 = "The food isn't good"
> let str3 = 'the2 cats'
> let str4 = 'switch on the light'

> const p1 =        // add your solution here
> const p2 =        // add your solution here

> p1.test(str1) || p2.test(str1)
< true
> p1.test(str2) || p2.test(str2)
< false
> p1.test(str3) || p2.test(str3)
< false
> p1.test(str4) || p2.test(str4)
< true
```

b) For the given input string, change only whole word `red` to `brown`

```
> let s1 = 'bred red spread credible red;'

> s1.replace()      // add your solution here
< "bred brown spread credible brown;"
```

c) For the given input array, filter all elements that contains `42` surrounded by word characters.

```
> let a1 = ['hi42bye', 'nice1423', 'bad42', 'cool_42a', 'fake4b']

> a1.filter()       // add your solution here
< ["hi42bye", "nice1423", "cool_42a"]
```

d) For the given input list, filter all elements that start with `den` or end with `ly`

```
> let a2 = ['lovely', '1\ndentist', '2 lonely', 'eden', 'fly away', 'dent']

> a2.filter()        // add your solution here
< ["lovely", "2 lonely", "dent"]
```

e) For the given input string, change whole word `mall` to `1234` only if it is at the start of line.

```
> let para = `ball fall wall tall
mall call ball pall
wall mall ball fall
mallet wallet malls`

> console.log(para.replace())        // add your solution here
< ball fall wall tall
  1234 call ball pall
  wall mall ball fall
  mallet wallet malls
```

Alternation and Grouping

Many a times, you want to check if the input string matches more than one pattern. For example, whether a car color is *green* or *blue* or *red*. In programming terms, you need to perform OR conditional. This chapter will discuss various ways to build a single regexp to match an input string against multiple conditions.

OR conditional

In a conditional expression, you can use the logical operators to combine multiple conditions. With regular expressions, the `|` metacharacter is similar to logical OR. The regexp will match if any of the expression separated by `|` is satisfied. Each of these alternations is a full regexp. For example, anchors are specific to that particular alternation.

```
// match either 'cat' or 'dog'
> const pets = /cat|dog/
> pets.test('I like cats')
< true
> pets.test('I like dogs')
< true
> pets.test('I like parrots')
< false

// replace either 'cat' at start of string or 'cat' at end of word
> 'catapults concatenate cat scat'.replace(/^cat|cat\b/g, 'X')
< "Xapults concatenate X sX"

// replace either 'cat' or 'dog' or 'fox' with 'mammal'
> 'cat dog bee parrot fox'.replace(/cat|dog|fox/g, 'mammal')
< "mammal mammal bee parrot mammal"
```

Grouping

Often, there are some common things among the regexp alternatives. It could be common characters or regexp qualifiers like the anchors. In such cases, you can group them using a pair of parentheses metacharacters. Similar to `a(b+c)d = abd+acd` in maths, you get `a(b|c)d = abd|acd` in regular expressions.

```
// without grouping
> 'red reform read arrest'.replace(/reform|rest/g, 'X')
< "red X read arX"
// with grouping
> 'red reform read arrest'.replace(/re(form|st)/g, 'X')
< "red X read arX"

// without grouping
> 'par spare part party'.replace(/\bpar\b|\bpart\b/g, 'X')
< "X spare X party"
```

```
// taking out common anchors
> 'par spare part party'.replace(/\b(par|part)\b/g, 'X')
< "X spare X party"
// taking out common characters as well
// you'll later learn a better technique instead of using empty alternate
> 'par spare part party'.replace(/\bpar(|t)\b/g, 'X')
< "X spare X party"
```

Precedence rules

There's some tricky situations when using alternation. If it is used for testing a match to get `true/false` against a string input, there is no ambiguity. However, for other things like string replacement without `g` flag, it depends on a few factors. Say, you want to replace either `are` or `spared` — which one should get precedence? The bigger word `spared` or the substring `are` inside it or based on something else?

The regexp alternative which matches earliest in the input string gets precedence.

```
> let words = 'lion elephant are rope not'

// starting index of 'on' < index of 'ant' for given string input
// so 'on' will be replaced irrespective of order of alternations
> words.replace(/on|ant/, 'X')
< "liX elephant are rope not"
> words.replace(/ant|on/, 'X')
< "liX elephant are rope not"
```

So, what happens if two or more alternatives match on same index? The precedence is then left to right in the order of declaration.

```
> let mood = 'best years'

// starting index for 'year' and 'years' will always be same
// so, which one gets replaced depends on the order of alternations
> mood.replace(/year|years/, 'X')
< "best Xs"
> mood.replace(/years|year/, 'X')
< "best X"
```

Another example with `replace` to drive home the issue.

```
> let sample = 'ear xerox at mare part learn eye'

// this is going to be same as: replace(/ar/g, 'X')
> sample.replace(/ar|are|art/g, 'X')
< "eX xerox at mXe pXt leXn eye"
// this is going to be same as: replace(/are|ar/g, 'X')
> sample.replace(/are|ar|art/g, 'X')
< "eX xerox at mX pXt leXn eye"
// phew, finally this one works as expected
```

```
> sample.replace(/are|art|ar/g, 'X')
< "eX xerox at mX pX leXn eye"
```

Cheatsheet and Summary

Note	Description
`pat1|pat2|pat3`	multiple regexp combined as OR conditional
	each alternative can have independent anchors
`()`	group pattern(s)
`a(b|c)d`	same as `abd|acd`
Alternation precedence	pattern which matches earliest in the input gets precedence
	tie-breaker is left to right if patterns have same starting location

So, this chapter was about specifying one or more alternate matches within the same regexp using `|` metacharacter. Which can further be simplified using `()` grouping if there are common aspects. Among the alternations, earliest matching pattern gets precedence. Left to right ordering is used as a tie-breaker if multiple alternations match starting from the same location. In the next chapter, you'll learn how to construct alternation from an array of strings taking care of precedence rules. Grouping has various other uses too, which will be discussed in coming chapters.

Exercises

a) For the given input list, filter all elements that start with `den` or end with `ly`

```
> let a1 = ['lovely', '1\ndentist', '2 lonely', 'eden', 'fly away', 'dent']

> a1.filter()        // add your solution here
< ["lovely", "2 lonely", "dent"]
```

b) For the given input strings, replace all occurrences of `removed` or `reed` or `received` or `refused` with `X`.

```
> let s1 = 'creed refuse removed read'
> let s2 = 'refused reed redo received'

> const p1 =        // add your solution here

> s1.replace(p1, 'X')
< "cX refuse X read"
> s2.replace(p1, 'X')
< "X X redo X"
```

c) For the given input strings, replace `late` or `later` or `slated` with `A`.

```
> let str1 = 'plate full of slate'
> let str2 = "slated for later, don't be late"
```

```
> const p2 =          // add your solution here

> str1.replace(p2, 'A')
< "pA full of sA"
> str2.replace(p2, 'A')
< "A for A, don't be A"
```

Escaping metacharacters

You have seen a few metacharacters and escape sequences that help to compose a RegExp literal. There's also the `/` character used as a delimiter for RegExp objects. This chapter will discuss how to remove the special meaning of such constructs. Also, you'll learn how to take care of special characters when you are building a RegExp literal from normal strings.

To match the metacharacters literally, i.e. to remove their special meaning, prefix those characters with a `\` character. To indicate a literal `\` character, use `\\`.

```
// even though ^ is not being used as anchor, it won't be matched literally
> /b^2/.test('a^2 + b^2 - C*3')
< false
// escaping will work
> /b\^2/.test('a^2 + b^2 - C*3')
< true


> '(a*b) + c'.replace(/\(|\)/g, '')
< "a*b + c"


> '\\learn\\by\\example'.replace(/\\/g, '/')
< "/learn/by/example"
```

Dynamically escaping metacharacters

When you are defining the regexp yourself, you can manually escape the metacharacters where needed. However, if you have strings obtained from elsewhere and need to match the contents literally, you'll have to somehow escape all the metacharacters while constructing the regexp. The solution of course is to use regular expressions! Usually, the programming language itself would provide an in-built method for such cases. JavaScript doesn't, but MDN: Regular Expressions doc has it covered in the form of a function as shown below.

```
> function escapeRegExp(string) {
    return string.replace(/[.*+?^${}()|[\]\\]/g, '\\$&')
  }
```

There are many things in the above regexp that you haven't learnt yet. They'll be discussed in coming chapters. For now, it is enough to know that this function will automatically escape all the metacharacters. Examples are shown below.

```
// sample input on which regexp will be applied
> let eqn = 'f*(a^b) - 3*(a^b)'
// sample string obtained from elsewhere which needs to be matched literally
> const usr_str = '(a^b)'

// case 1: replace all matches
// escaping metacharacters using 'escapeRegExp' function
> const pat = new RegExp(escapeRegExp(usr_str), 'g')
> pat
< /\(a\^b\)/g
```

```
> eqn.replace(pat, 'c')
< "f*c - 3*c"


// case 2: replace only at the end of string
> eqn.replace(new RegExp(escapeRegExp(usr_str) + '$'), 'c')
"f*(a^b) - 3*c"
```

Dynamically building alternation

Examples in previous chapter showed cases where a single regexp can contain multiple patterns combined using `|` metacharacter. Often, you have an array of strings and you need to match any of their content literally. To do so, you need to escape all the metacharacters before combining the strings with `|` metacharacter. The function shown below uses the `escapeRegExp` function introduced in the previous section.

```
> function unionRegExp(arr) {
      return arr.map(w => escapeRegExp(w)).join('|')
  }
```

And here's some examples with `unionRegExp` function used to construct the required regexp.

```
// here, order of alternation wouldn't matter
// and assume that other regexp features aren't needed
> let w1 = ['c^t', 'dog$', 'f|x']
> const p1 = new RegExp(unionRegExp(w1), 'g')
> p1
< /c\^t|dog\$|f\|x/g
> 'c^t dog$ bee parrot f|x'.replace(p1, 'mammal')
< "mammal mammal bee parrot mammal"


// here, alternation precedence rules needs to be applied first
// and assume that the terms have to be matched as whole words
> let w2 = ['hand', 'handy', 'handful']
// sort by string length, longest first
> w2.sort((a, b) => b.length - a.length)
< ["handful", "handy", "hand"]
> const p2 = new RegExp(`\\b(${unionRegExp(w2)})\\b`, 'g')
> p2
< /\b(handful|handy|hand)\b/g
// note that 'hands' and 'handed' isn't replaced
> 'handful handed handy hands hand'.replace(p2, 'X')
< "X handed X hands X"
```

    ⓘ The XRegExp utility provides XRegExp.escape and XRegExp.union methods. The union method has additional functionality of allowing a mix of string and RegExp literals and also takes care of renumbering backreferences.

source and flags properties

If you need the contents of a RegExp object, you can use `source` and `flags` properties to get the pattern string and flags respectively. These methods will help you to build a RegExp object using contents of another RegExp object.

```
> const p3 = /\bpar\b/
> const p4 = new RegExp(p3.source + '|cat', 'g')

> p4
< /\bpar\b|cat/g
> console.log(p4.source)
< \bpar\b|cat
> p4.flags
< "g"


> 'cater cat concatenate par spare'.replace(p4, 'X')
< "Xer X conXenate X spare"
```

Escaping delimiter

Another character to keep track for escaping is the delimiter used to define the RegExp literal. Or depending upon the pattern, you can use the `new RegExp` constructor to avoid escaping.

```
> let path = '/foo/123/foo/baz/ip.txt'

// this is known as 'leaning toothpick syndrome'
> path.replace(/^\/foo\/123\//, '~/')
< "~/foo/baz/ip.txt"

// using 'new RegExp' improves readability and can reduce typos
> path.replace(new RegExp(`^/foo/123/`), '~/')
< "~/foo/baz/ip.txt"
```

Cheatsheet and Summary

Note	Description
`\`	prefix metacharacters with `\` to match them literally
`\\`	to match `\` literally
`source`	property to convert RegExp object to string
	helps to insert a RegExp inside another RegExp
`flags`	property to get flags of a RegExp object
`RegExp(`pat`)`	helps to avoid or reduce escaping the `/` delimiter character
Alternation precedence	tie-breaker is left to right if patterns have same starting location
	robust solution: sort the alternations based on length, longest first

Exercises

a) Transform given input strings to expected output using same logic on both strings.

```
> let str1 = '(9-2)*5+qty/3'
> let str2 = '(qty+4)/2-(9-2)*5+pq/4'

> const p1 =         // add your solution here
> str1.replace(p1, '35')
< "35+qty/3"
> str2.replace(p1, '35')
< "(qty+4)/2-35+pq/4"
```

b) Replace any matching item from given array with X for the given input strings.

```
> let items = ['a.b', '3+n', 'x\\y\\z', 'qty||price', '{n}']
> const p2 =         // add your solution here

> '0a.bcd'.replace(p2, 'X')
< "0Xcd"
> 'E{n}AMPLE'.replace(p2, 'X')
< "EXAMPLE"
> '43+n2 ax\\y\\ze'.replace(p2, 'X')
< "4X2 aXe"
```