learnbyexample

UNDERSTANDING JAVASCRIPT REGEXP

- 300+ examples
- 100+ exercises

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 understand Regular Expressions, a mini-programming language for all sorts of text processing needs.

This 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. Make an effort to understand the sample input as well as the solution presented and check if the output changes (or not!) when you alter some part of the input and the code. As an analogy, consider learning to drive a car — no matter how much you read about them or listen to explanations, you'd need practical experience to become proficient.

Prerequisites

You should be familiar with programming basics. You should also have a working knowledge of JavaScript syntax and functional programming concepts like `map` and `filter` .

You are also expected to get comfortable with reading documentation, searching online, visiting external links provided for further reading, tinkering with illustrated examples, asking for help when you are stuck and so on. In other words, be proactive and curious instead of just consuming the content passively.

Conventions

- The examples presented here have been tested on the Chrome/Chromium console and includes features not available in other browsers and platforms.
- Code snippets shown are copy pasted from the console and modified for presentation purposes. Some of the 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 throughout the book for you to explore certain topics in more depth.
- The learn_js_regexp repo has all the code snippets, exercises and other details related to the book. If you are not familiar with the `git` command, click the **Code** button on the webpage to get the files.

Acknowledgements

- MDN: Regular Expressions — documentation and examples
- /r/learnjavascript/ and /r/regex/ — helpful forums for beginners and experienced programmers alike
- stackoverflow — for getting answers to pertinent questions on JavaScript and regular expressions
- tex.stackexchange — for help on pandoc and `tex` related questions
- canva — cover image
- Warning and Info icons by Amada44 under public domain
- oxipng, pngquant and svgcleaner — optimizing images

Feedback and Errata

I would highly appreciate it if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: https://github.com/learnbyexample/learn_js_regexp/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at https://github.com/learnbyexample.

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

License

Book version

2.0

See Version_changes.md to track changes across book versions.

Why is it needed?

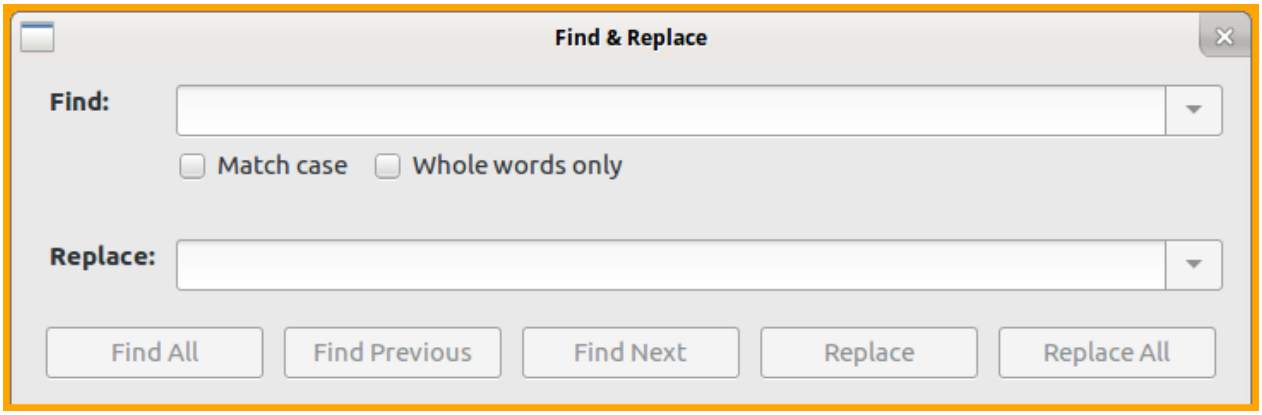
Regular Expressions is a versatile tool for text processing. You'll find them included as part of the standard library of most programming languages that are used for scripting purposes. If not, you can usually find a third-party library. Syntax and features of regular expressions vary from language 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 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, repetition and so on.

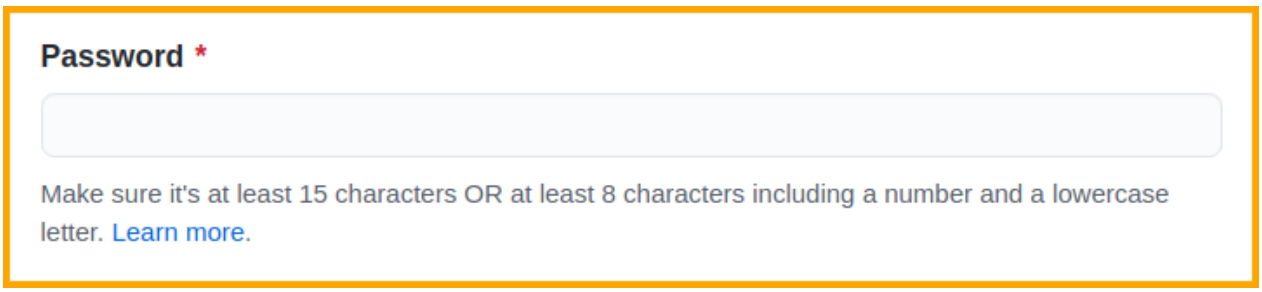
Here are some common use cases:

- 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, digits, 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.

You are likely to be familiar with graphical search and replace tools, like the screenshot shown below from LibreOffice Writer. **Match case**, **Whole words only**, **Replace** and **Replace All** are some of the basic features supported by regular expressions.



Another real world use case is password validation. The screenshot below is from GitHub sign up page. Performing multiple checks like **string length** and the **type of characters allowed** is another core feature of regular expressions.



Here are some articles on regular expressions to know about its history and the type of prob-

lems it is suited for.

- 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 — demystifies the often (mis)quoted meme
- wikipedia: Regular expression — this article includes discussion on regular expressions as a formal language as well as details about various implementations

How this book is organized

This book introduces concepts one by one and exercises at the end of chapters will require only the features introduced until that chapter. Each concept is accompanied by plenty of examples to cover multiple problems and corner cases. As mentioned before, it is highly recommended that you follow along the examples by typing out the code snippets manually. It is important to understand both the nature of the sample input string as well as the actual programming command used. There are two interlude chapters that give an overview of useful tools and some more resources are collated in the final chapter.

- RegExp introduction
- Anchors
- Alternation and Grouping
- Escaping metacharacters
- Dot metacharacter and Quantifiers
- Interlude: Tools for debugging and visualization
- Working with matched portions
- Character class
- Groupings and backreferences
- Interlude: Common tasks
- Lookarounds
- Unicode
- Further Reading

By the end of the book, you should be comfortable with both writing and reading regular expressions, how to debug them and know when to *avoid* them.

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 the 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.

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

Console and documentation

As mentioned in the Preface chapter, examples presented in this book have been tested on the 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 reused across different chapters — open another tab in such cases to avoid errors.

See MDN: Regular Expressions Guide and MDN: Regular Expressions Reference for examples, documentation and browser 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 `//` delimiters.

```
> 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 object
> /is/.test(sentence)
< true
> /z/.test(sentence)
< false
```

Here are 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 are 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 of the elements contain 'stat'
> words.some(w => /stat/.test(w))
< false
```

Flags

Some of the regular expressions functionality is enabled by passing flags, represented by a lowercase letter. Flags are similar to command line options, for example `grep -i` performs case insensitive matching.

In this chapter, the following flags will be discussed:

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

Examples for the `i` flag are shown below. `g` flag will be discussed in the 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, enables 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 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 the `//` format is the ability to dynamically construct the regexp. For example, to insert the 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/
```

replace() method

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

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

// change all the 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 the 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 if 'word' has to be changed
> word = word.replace(/cat/, 'hack')
< 'hacker'
> word
< 'hacker'
```

The use of the `g` flag with the `test()` method allows some additional functionality. See MDN: test for examples. However, in my opinion, it is easy to fall into a habit of using `g` with `test()` and get undesired results. Instead, I'd suggest to use the match() method and explicitly write the required logic instead of relying on the `g` flag.

Cheatsheet and Summary

Note	Description
MDN: Regular Expressions	MDN reference 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 the pattern is present anywhere in the 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 optional 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 the `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. From the next chapter onwards, you'll learn regular expression syntax and features.

Exercises

Try to solve the exercises in every chapter using only the features discussed until that chapter. Some of the exercises will be easier to solve with techniques presented in the later chapters, but the aim of these exercises is to explore the features presented so far.

All the exercises are also collated together in one place at Exercises.md. For solutions, see Exercise_solutions.md.

1) 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 pat1 =      // add your solution here

> pat1.test(s1)
< true
> pat1.test(s2)
< false
> pat1.test(s3)
< true
```

2) For the given array, filter all elements that do *not* contain `e` .

```
> let items = ['goal', 'new', 'user', 'sit', 'eat', 'dinner']

> items.filter(w => test(w))       // add your solution here
< ['goal', 'sit']
```

3) Replace only the first occurrence of `5` with `five` for the given string.

```
> let ip = 'They ate 5 apples and 5 oranges'

> ip.replace()       // add your solution here
< 'They ate five apples and 5 oranges'
```

4) Replace all occurrences of `5` with `five` for the given string.

```
> let ip = 'They ate 5 apples and 5 oranges'

> ip.replace()      // add your solution here
< 'They ate five apples and five oranges'
```

5) Replace all occurrences of `note` irrespective of case with `X` .

```
> let ip = 'This note should not be NoTeD'

> ip.replace()     // add your solution here
< 'This X should not be XD'
```

6) 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(e => test(e))       // add your solution here
                       .join('\n'))
< items qty
  mango 50
  onion 31
  water 10
```


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

7) For the given array, filter all elements that contain either `a` or `w` .

```
> let items = ['goal', 'new', 'user', 'sit', 'eat', 'dinner']

> items.filter(w => test(w) || test(w))     // add your solution here
< ['goal', 'new', 'eat']
```

8) For the given array, filter all elements that contain both `e` and `n` .

```
> let items = ['goal', 'new', 'user', 'sit', 'eat', 'dinner']

> items.filter(w => test(w) && test(w))     // add your solution here
< ['new', 'dinner']
```

9) For the given string, replace `0xA0` with `0x7F` and `0xC0` with `0x1F` .

```
> let ip = 'start address: 0xA0, func1 address: 0xC0'

> ip.replace()        // add your solution here
< 'start address: 0x7F, func1 address: 0x1F'
```

Anchors

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

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 `\` character (discussed in the Escaping metacharacters chapter).

String anchors

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

```
// ^ is placed as a prefix to the search term
> /^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 matching to the end of string, the `$` metacharacter is used.

```
// $ is placed as a suffix to the search term
> /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. The effect is similar to comparing strings using the `==` operator.

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

You can emulate string concatenation operations by using the anchors by themselves as a pattern.

```
// insert text at the start of a string
> 'live'.replace(/^/, 're')
< 'relive'
> 'send'.replace(/^/, 're')
< 'resend'

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

Line anchors

A string input may contain single or multiple lines. The characters `\r`, `\n`, `\u2028` (line separator) and `\u2029` (paragraph separator) are considered as line separators. 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

// filter elements having lines ending with 'are'
> let elements = ['spare\ntool', 'par\n', 'dare', 'spared']
> elements.filter(e => /are$/m.test(e))
< ['spare\ntool', 'dare']

// check if any whole 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 line separator character at the end of string, there is an additional start/end of line match after the separator.

```
// 'fig ' is inserted three times
> console.log('1\n2\n'.replace(/^/mg, 'fig '))
< fig 1
  fig 2
  fig

> console.log('1\n2\n'.replace(/$/mg, ' apple'))
< 1 apple
  2 apple
   apple
```

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

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 just 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 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 words = 'par spar apparent spare part'

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

Using word boundary as a pattern by itself can yield creative solutions:

```
// space separated words to double quoted csv
// note that the 'replace' method is used twice here
> let words = 'par spar apparent spare part'
> console.log(words.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
> 'output=num1+35*42/num2'.replace(/\b/g, ' ')
< ' output = num1 + 35 * 42 / num2 '
// excess space at the start/end of string can be trimmed off
// later you'll learn how to add a qualifier so that trim is not needed
> 'output=num1+35*42/num2'.replace(/\b/g, ' ').trim()
< 'output = num1 + 35 * 42 / num2'
```

Opposite Word Anchor

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, you might end up matching things you didn't intend!

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

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

Here are 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 the start of string
`$`	restricts the match to the end of string
`m`	flag to match the start/end of line with `^` and `$` anchors
	`\r` , `\n` , `\u2028` and `\u2029` are line separators
	DOS-style files use `\r\n` , may need special attention
`\b`	restricts the match to the start and 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 other string methods and expressions. Practice, experience and imagination would help you construct creative solutions. In the coming chapters, you'll see examples for anchors in combination with other regexp features.

Exercises

1) 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 pat1 =      // add your solution here
> const pat2 =      // add your solution here

> pat1.test(str1) || pat2.test(str1)
< true
> pat1.test(str2) || pat2.test(str2)
< false
> pat1.test(str3) || pat2.test(str3)
< false
> pat1.test(str4) || pat2.test(str4)
< true
```

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

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

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

3) For the given array, filter all elements that contain `42` surrounded by word characters.

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

> items.filter(e => test(e))       // add your solution here
< ['hi42bye', 'nice1423', 'cool_42a']
```

4) For the given input array, filter all elements that start with `den` or end with `ly` .

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

> items.filter(e => test(e) || test(e))        // add your solution here
< ['lovely', '2 lonely', 'dent']
```

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

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

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

6) For the given array, filter all elements having a line starting with `den` or ending with `ly` .

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

> items.filter(e => test(e) || test(e))      // add your solution here
< ['lovely', '1\ndentist', '2 lonely', 'fly\nfar', 'dent']
```

7) For the given input array, filter all whole elements `12\nthree` irrespective of case.

```
> let items = ['12\nthree\n', '12\nThree', '12\nthree\n4', '12\nthree']

> items.filter(e => test(e))     // add your solution here
< ['12\nThree', '12\nthree']
```

8) For the given input array, replace `hand` with `X` for all elements that start with `hand` followed by at least one word character.

```
> let items = ['handed', 'hand', 'handy', 'un-handed', 'handle', 'hand-2']

> items.map(w => w.replace())        // add your solution here
< ['Xed', 'hand', 'Xy', 'un-handed', 'Xle', 'hand-2']
```

9) For the given input array, filter all elements starting with `h` . Additionally, replace `e` with

`X` for these filtered elements.

```
> let items = ['handed', 'hand', 'handy', 'unhanded', 'handle', 'hand-2']

> items.filter(w => test(w)).map(w => w.replace())        // add your solution here
< ['handXd', 'hand', 'handy', 'handlX', 'hand-2']
```

10) Why does the following code show `false` instead of `true` ?

```
> /end$/.test('bend it\nand send\n')
< false
```

Alternation and Grouping

Many a times, you want to check if the input string matches multiple patterns. For example, whether a product's color is *green* or *blue* or *red*. This chapter will show how to use alternation for such cases. These patterns can have some common elements between them, in which case grouping helps to form terser regexps. This chapter will also discuss the precedence rules used to determine which alternation wins.

Alternation

A conditional expression combined with logical OR evaluates to `true` if any of the conditions is satisfied. Similarly, in regular expressions, you can use the `|` metacharacter to combine multiple patterns to indicate logical OR. The matching will succeed if any of the alternate patterns is found in the input string. These alternatives have the full power of a regular expression, for example they can have their own independent anchors. Here are some examples.

```
// 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 the start of string or 'cat' at the end of word
> 'catapults concatenate cat scat cater'.replace(/^cat|cat\b/g, 'X')
< 'Xapults concatenate X sX cater'
// replace 'cat' or 'dog' or 'fox' with 'mammal'
> 'cat dog bee parrot fox'.replace(/cat|dog|fox/g, 'mammal')
< 'mammal mammal bee parrot mammal'
```

You might infer from the above examples that there can be situations where many alternations are required. See the Dynamically building alternation section for examples and details.

Grouping

Often, there are some common portions among the regexp alternatives. It could be common characters, qualifiers like the anchors and so on. 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 alternates
> 'par spare part party'.replace(/\bpar(|t)\b/g, 'X')
< 'X spare X party'
```


There are many more uses for grouping than just forming a terser regexp. They will be discussed as they become relevant in the coming chapters.

Precedence rules

There are tricky situations when using alternation. There is no ambiguity if it is used to get a boolean result by testing a match against a string input. However, for cases like string replacement, 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 the given string input
// so 'on' will be replaced irrespective of the order of alternations
> words.replace(/on|ant/, 'X')
< 'liX elephant are rope not'
> words.replace(/ant|on/, 'X')
< 'liX elephant are rope not'
```

What happens if the alternatives have the same starting index? The precedence is left-to-right in the order of declaration.

```
> let mood = 'best years'

// starting index for 'year' and 'years' will always be the 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'
```

Here's another example 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 needed
> 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 conditional OR 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 matches have the same starting location

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

Exercises

1) For the given input array, filter all elements that start with `den` or end with `ly` .

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

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

2) For the given array, filter all elements having a line starting with `den` or ending with `ly` .

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

> items.filter()        // add your solution here
< ['lovely', '1\ndentist', '2 lonely', 'fly\nfar', 'dent']
```

3) 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 pat1 =      // add your solution here

> s1.replace(pat1, 'X')
< 'cX refuse X read'
> s2.replace(pat1, 'X')
< 'X X redo X'
```

4) 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 pat2 =      // add your solution here

> str1.replace(pat2, 'A')
< 'pA full of sA'
> str2.replace(pat2, 'A')
< "A for A, don't be A"
```

Escaping metacharacters

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

Escaping with backslash

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

```
// even though ^ is not being used as an 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

// match ( or ) literally
> '(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 a builtin method for such cases. JavaScript doesn't, but MDN: Regular Expressions Guide 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 the 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 the 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 the '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 the input string
> eqn.replace(new RegExp(escapeRegExp(usr_str) + '$'), 'c')
"f*(a^b) - 3*c"
```

Note that the `/` delimiter character isn't escaped in the above function. You can use `[.*+?^${}()|[\]\\\/]` to escape the delimiter as well.

Dynamically building alternation

Examples in the previous chapter showed cases where a single regexp can contain multiple patterns combined using the `|` metacharacter. Often, you have an array of strings and the requirement is to match any of the elements literally. To do so, you need to escape all the metacharacters before combining the strings with the `|` 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 are some examples with the `unionRegExp()` function used to construct the required regexp.

```
// here, the 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 the 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' aren't replaced
> 'handful handed handy hands hand'.replace(p2, 'X')
< 'X handed X hands X'
```

The XRegExp utility provides handy methods like XRegExp.escape() and XRegExp.union(). 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 the `source` and `flags` properties to get the pattern string and flags respectively. These methods will help you to build a RegExp object using the 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 the 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 = '/home/joe/report/sales/ip.txt'

// this is known as 'leaning toothpick syndrome'
> path.replace(/^\/home\/joe\//, '~/')
< '~/report/sales/ip.txt'

// using 'new RegExp' improves readability and can reduce typos
> path.replace(new RegExp(`^/home/joe/`), '~/')
< '~/report/sales/ip.txt'
```

Escape sequences

Certain characters like tab and newline can be expressed using escape sequences as `\t` and `\n` respectively. These are similar to how they are treated in normal string literals. However, `\b` is for word boundaries as seen earlier, whereas it stands for the backspace character in normal string literals.

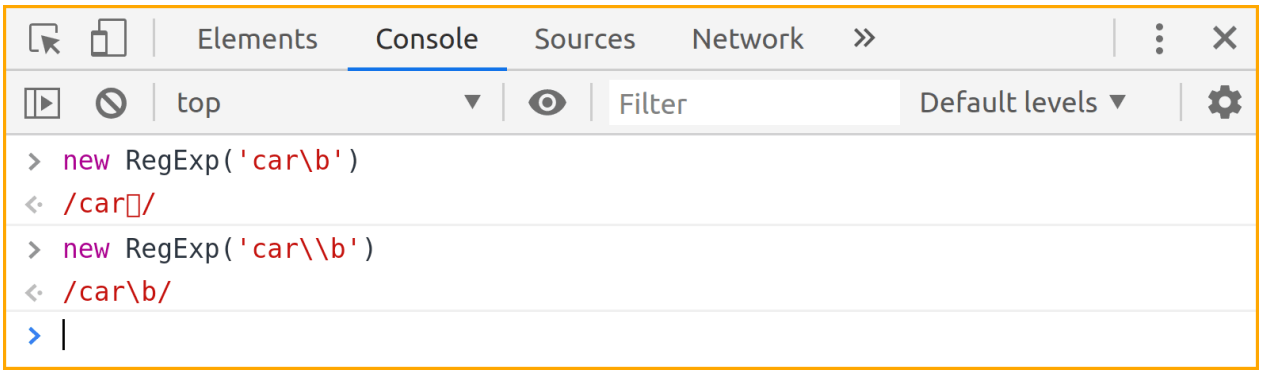
Additionally, there are several sequences that are specific to regexps. The full list is mentioned in the **Using special characters** section of MDN documentation. These are `\b \B \cX \d \D \f \k<name> \n \p \P \r \s \S \t \uhhhh \u{hhhh} \v \w \W \xhh \0` . Here are some examples:

```
> 'a\tb\tc'.replace(/\t/g, ':')
< 'a:b:c'

> '1\n2\n3'.replace(/\n/g, ' ')
< '1 2 3'

// use \\ instead of \ when constructing regexp from string literals
// when you need to represent a single backslash character literally
> new RegExp('123\tabc')
< /123    abc/
> new RegExp('123\\tabc')
< /123\tabc/
```

Here's a console screenshot of another example.



If an escape sequence is not defined, it will be treated as the character it escapes.

```
// here \e is treated as e
> /\e/.test('hello')
< true
```

You can also represent a character using hexadecimal escape of the format `\xhh` where `hh` are exactly two hexadecimal characters. If you represent a metacharacter using escapes, it will be treated literally instead of its metacharacter feature. The Codepoints section will discuss escapes for unicode characters.

```
// \x20 is the space character
> 'h e l l o'.replace(/\x20/g, '')
< 'hello'

// \x7c is the '|' character
// but it won't be treated as the alternation metacharacter
> '12|30'.replace(/2\x7c3/g, '5')
< '150'
> '12|30'.replace(/2|3/g, '5')
< '15|50'
```


See ASCII code table for a handy cheatsheet with all the ASCII characters and their hexadecimal representations.

Cheatsheet and Summary

Note	Description
`\`	prefix metacharacters with `\` to match them literally
`\\`	to match `\` literally
`source`	property to convert a RegExp object to a 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 matches have the same starting location
	robust solution: sort the alternations based on length, longest first
`\t`	escape sequences like those supported in string literals
`\b`	word boundary in regexps but backspace in string literals
`\e`	undefined escapes will match the character it escapes
`\xhh`	represent a character using hexadecimal values
`\x7c`	matches `|` literally

Exercises

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

```
> let str1 = '(9-2)*5+qty/3-(9-2)*7'
> let str2 = '(qty+4)/2-(9-2)*5+pq/4'

> const pat1 =      // add your solution here
> str1.replace()        // add your solution here
< '35+qty/3-(9-2)*7'
> str2.replace()        // add your solution here
< '(qty+4)/2-35+pq/4'
```

2) Replace `(4)\|` with `2` only at the start or end of the given input strings.

```
> let s1 = '2.3/(4)\\|6 fig 5.3-(4)\\|'
> let s2 = '(4)\\|42 - (4)\\|3'
> let s3 = 'two - (4)\\|\n'

> const pat2 =      // add your solution here

> s1.replace()      // add your solution here
< '2.3/(4)\\|6 fig 5.3-2'
> s2.replace()      // add your solution here
< '242 - (4)\\|3'
> s3.replace()      // add your solution here
< 'two - (4)\\|\n'
```

3) Replace any matching element from the array `items` with `X` for given the input strings. Match the elements from `items` literally. Assume no two elements of `items` will result in any matching conflict.

```
> let items = ['a.b', '3+n', 'x\\y\\z', 'qty||price', '{n}']

// add your solution here
> const pat3 =      // add your solution here

> '0a.bcd'.replace(pat3, 'X')
< '0Xcd'
> 'E{n}AMPLE'.replace(pat3, 'X')
< 'EXAMPLE'
> '43+n2 ax\\y\\ze'.replace(pat3, 'X')
< '4X2 aXe'
```

4) Replace the backspace character `\b` with a single space character for the given input string.

```
> let ip = '123\b456'

> ip.replace()      // add your solution here
< '123 456'
```

5) Replace all occurrences of `\e` with `e` .

```
> let ip = 'th\\er\\e ar\\e common asp\\ects among th\\e alt\\ernations'

> ip.replace()      // add your solution here
< 'there are common aspects among the alternations'
```

6) Replace any matching item from the array `eqns` with `X` for given the string `ip` . Match the items from `eqns` literally.

```
> let ip = '3-(a^b)+2*(a^b)-(a/b)+3'
> let eqns = ['(a^b)', '(a/b)', '(a^b)+2']

// add your solution here
> const pat4 =      // add your solution here

> ip.replace(pat4, 'X')
< '3-X*X-X+3'
```