Introducing Regular Expressions

JavaScript and TypeScript

Jörg Krause

Apress®

Jörg Krause

Introducing Regular Expressions

JavaScript and TypeScript

Jörg Krause
Berlin, Germany

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com . For detailed information about how to locate your book's source code, go to www.apress.com/source-code/ .

ISBN 978-1-4842-2507-3 e-ISBN 978-1-4842-2508-0
DOI 10.1007/978-1-4842-2508-0

Library of Congress Control Number: 2016962076

the material contained herein.

Printed on acid-free paper

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

Introduction

This book explains regular expressions in JavaScript. Such expressions are search pattern – instructions to search within text. This can be really complex but even powerful.

> *JavaScript vs. ECMAScript*
> I use the name JavaScript in this book, although it's officially called ECMAScript. The book is based on the version ES5. Because JavaScript is valid TypeScript and ECMAScript 6 (ES6) is compatible with ES5, all code snippets in this book will run using TypeScript or new versions ov JavaScript, too.

Who Should Read this Book?

This book is aimed at beginners and web developers who are new to the Web. JavaScript mainly serves the front-end developers. Maybe you are also a web designer, who discovered JavaScript as an excellent way to upgrade your web pages with dynamic elements. Here, you are dealing with texts, forms, and the presentation of database content—that is, everything that constitutes a dynamic web site. You will need to know at least one of these, namely for the creation of a professional user interface, in particular a clear form.

In any case, I tried to avoid prerequisites or conditions for readers of this book. You do not need to be a computer scientist, you do not need perfect command of a particular language, don't need to know rocket science. No matter in the context in which you have encountered Jade, you will be able to understand this book. However, developers will get the most benefit from this book.

> **JavaScript**
> To understand all the examples, you need a working environment for executing scripts. This can take place in a browser, on a website, or by using Node.js.

What You Should Know

Readers of this series need hardly any requirements. Knowledge of some HTML

cannot harm, however, and it doesn’t hurt to already have seen a static HTML page (the source code, of course). I assume that you have access to a current operating system—on which there is an editor with which you can create web pages.

How to Read the Text

I will not dictate how you should read this text. In the first draft of the structure, I tried several variations and found that there is no ideal form. However, readers tend to consume smaller chunks of information in independent chapters with focused content. This book follows this trend by reducing information to small focused issues without the “blah-blah” to inflate the volume.

Beginners should read the text as a narrative from the first page to the last page. Those who are already somewhat familiar with the information can safely skip certain sections.

Conventions Used in the Book

The theme is not easy to master technically, because scripts are often too wide. It would be nice if you could support the best optical reading form. I have therefore included extra line breaks to aid readability, but that’s not required in your development environment’s editor.

In general, each program code is set to a non-proportional font. In addition, scripts have line numbers:

```
1   body {
2     color: black;
3   }
```

If you need to enter something in the prompt or in a dialog box, this part of the statement is in bold:

```
$ bower install bootstrap
```

The first character is the prompt, which is not entered. In the book, I use the Linux prompt from the bash shell. The commands will work, without exception, unchanged—even on Windows. The only difference is the `C:>` command prompt or something similar at the beginning of a line. Usually, the instructions are related to relative paths or no paths at all, so the actual prompt shouldn’t matter, despite the fact that you are in your working folder.

Expressions and command lines are sometimes peppered with all types of characters; in almost all cases, this depends on each character. Often, I’ll discuss

the use of certain characters in precisely such an expression. Then the “important” characters have line breaks to set them apart. They also have line numbers, which are used to reference the affected symbol in the text exactly (note the : (colon) character in line 2):

```
1   a.test {
2     :hover {
3       color: red
4     }
5   }
```

The font is non-proportional so that the characters are countable. Opening and closing parentheses are always by themselves.

Symbols

To facilitate the orientation in the search for a solution, there is a whole range of symbols that are used in the text.

Tip

This is a tip.

Information

This is information.

Warning

This is a warning.

Contents

Contents at a Glance

About the Author

Jörg Krause

has been working with software and software technology since the early 1980s, beginning with a ZX 81 and taking his first steps as a programmer in BASIC and assembly language. He studied information technology at Humboldt University, Berlin, but left early, in the 1990s, to start his own company. He has worked with Internet technology and software development since the early days, when CompuServe and FidoNet dominated. He's been with Microsoft technologies and software since Windows 95.

In 1998, he worked on one of the first commercial e-commerce solutions, and wrote his first book in Germany, *eCommerce and Online Marketing* , published by Carl Hanser Verlag, Munich. Due to the book's wide success, he began working as a freelance consultant and author to share his experience and knowledge with others. He has written several books for major publishers such as Apress, Hanser, Addison Wesley, and others, as well as several self-published books—for a total of over 60 titles. He also publishes articles in magazines and speaks at major conferences in Germany. Currently, Jörg works as an independent consultant, software developer, and author in Berlin, Germany.

In his occasional spare time, Jörg enjoys reading thrillers and science fiction novels, and going on a round of golf.

Jörg Krause, Introducing Regular Expressions, DOI 10.1007/978-1-4842-2508-0_1

1. Introducing Regular Expressions

Jörg Krause[1]

(1) Berlin, Germany

To get a head start on introducing regular expressions , I'll start with an example. It's one that you've experienced hundreds of times. When you enter customer data online, many web forms ask you for an email address. To avoid an incorrectly typed address, an immediate validation makes sense. One way would be to split the string in parts (before and after the `@` character), analyzing the positions of the dots and the number of characters after the last dot (that's the top-level domain).

After a few ifs and loops, you're almost done. Or you simply use a regular expression:

```
^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)* @ [a-zA-Z0-9-
]+\.([azA-Z]{2,3})$
```

Did you get that? If this is your first time with regular expressions , it is probably hard to read.

Regular expressions are a form of pattern recognition in regular text (read: string). Regular expressions compare the pattern with the text. The whole expression, encapsulated as an object in a script or programming language, would return either `true` or `false`. The result obviously tells the caller whether the comparison was successful or not. Hence, the expression can be better understood if you see it in the context of an actual language. Because this book is dedicated to JavaScript , the usage in that language would look like this:

```
1   var email = "joerg@krause.net";
2
```

```
    3   console.log(check(email));
    4
    5   function check(email) {
    6     if (email.match(/^[_a-zA-Z0-9-]+(\.[_a-zA-
Z0-9-]+)*@[a-zA-Z0-9-]+\\
    7   .([a-zA-Z]{2,3})$/)) {
    8         return true;
    9     } else {
   10         return false;
   11     }
   12   }
```

Here the expression is made by using typical literals as a boundary `/expression/` and the comparison is made by the `match` function that each string object provides. Be aware that the slashes must not be put in quotes. It's a literal that creates an object.

A Test Environment

For the first steps, it is helpful to use a JavaScript test console that is available online. I recommend using Repl.it.[1]

Figure 1-1. The example in Repl.it

REPL

The term REPL is an abbreviation for Read-Eval-Print-Loop. It's a method of working interactively with a script language. Read more about it on

Wikipedia.[2]

Copy or Scaffold?

I'm going to show many useful expressions in Chapter 4. This first chapter, however, shows both trivial and non-trivial expressions that are ready to use. Because the expressions are sometimes tricky and hard to read, you can download all examples.

Website

Visit this book's support website.[3]

If you want to become a professional JavaScript developer, regular expressions are a part of your toolset. You should try to understand the expressions completely and start creating your own.

How Does It Work?

You might be curious to know how the preceding expression works.

If you start analyzing such expressions, you'd best start with the extraction of special characters. These include one of the following in this particular expression: `^, $, +, *, ?, [], ()`. All other characters do not possess a special meaning here. Regular characters are a minority. Usually, such patterns use placeholders and descriptive characters more than actual letters in a word.

Here is an overview of the special characters :

- `^` lets the recognition start at the beginning. If you write `^x`, the expression will match the letter "x" only if it appears at the very first character.
- `$` lets you define where the pattern ends.
- `*` is a placeholder that means no or any number of characters.
- `+` is a placeholder that means one or any number of characters.
- `?` is a placeholder that means no or one character.
- `[a-z]` defines one character out of group of letters or digits. You can use uppercase letters, lowercase letters, or digits by simply placing them in the

brackets, or you can define them as a range, as shown in the example.

- `()` groups characters or strings of characters. You can use the set operators `*`, `+`, and `?` in such a group, too.
- `{}` is a repetition marker that defines the character before the braces. It can be repeated multiple times. The range can be defined by numbers; if the start and the end are given separately, the numbers are written with a comma (`{3,7}`).
- `\` (the backslash) masks metacharacters and special characters so that they do no longer possess a special meaning.
- `.` represents exactly one character. If you actually need a dot, just write `\.`.

Now you can easily split the expression quite well. The `@` character represents itself and the first step splits the expression:

```
1    ^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*
2    @
3    [a-zA-Z0-9-]+\.([a-zA-Z]{2,3})$
```

The part before the `@` character must have at least one character. This is forced by the first character definition `[_a-zA-Z0-9-]`, with all acceptable characters together with the + sign. Then, the expression might be followed by an actual dot `\.`, which by itself can be followed by one or more characters. The whole “dot plus more characters” group is optional and can be repeated endlessly (`*`). The second part is similar. The set definition is missing the underscore character, which may not appear in regular domain names. The dot is not optional (no set operator) and the remaining part can be two or three characters long (that’s ignoring the new domain names with four or more characters , but I think you get the idea).

Resolving Expressions

The expression might appear here, but it’s still far from being perfect.

There are a few cases where the expression may reject acceptable email addresses and also accept irregular ones. An expression is still hard to explain. Because you’ll be challenged with more complex examples soon, I’ll use another layout here:

```
   1   ^                          // Start at the
beginning
   2      [_a-zA-Z0-9-]           // Define a group
   3      +                       // One or more
times
   4        (                     // Group 1
   5            \.                // a "real" dot
   6            [_a-zA-Z0-9-]     // Another group
   7            +                 // One or more
times
   8        )                     // /* End group 1
*/
   9        *                     // Group optional
or multiple times
  10     @                        // The @ character
  11     [a-zA-Z0-9-]             // Character
definition
  12     +                        // On or multiple
times
  13     \.                       // A "real" dot
  14       (                      // Group 2
  15           [a-zA-Z]           // A character
definition
  16           {2,3}              // Two or three
characters
  17       )                      // /* End group 2
*/
  18   $                          // End of text
must come now
```

That’s a lot easier, isn’t it? Unfortunately, you can’t write it in that way in JavaScript . I’m going to use this form only to break down tricky expressions. If you struggle with an expression in this book, just try to resolve the pattern by putting each character on one line and write down what it means .

Footnotes

1 https://repl.it/languages/JavaScript

2 https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

3 http://www.joergkrause.de/?p=219

Jörg Krause, Introducing Regular Expressions, DOI 10.1007/978-1-4842-2508-0_2

2. Recognizing Patterns

Jörg Krause[1]

(1) Berlin, Germany

In this chapter, I'm going to show basic techniques using simple examples. This is probably not enough for daily use, but it's a good basis for your own experiments.

Basics

What happens if an expression is being evaluated? Actually it's always the goal to search a text portion within another, often larger, text. The text searched may come from a formula, a file, a database, or it may be just a string. But that's not the point. You can use a simple search function for that, and most often, such a simple search is more efficient. Regular expressions define properties within the search text. That way, you don't need to change the search term again and again, but give a range of variations. Search and replace may come to a whole new level.

Pattern

Regular expressions recognize a string with text by using a *pattern*.

When searching, an expression is used to resolve a comparison between strings. The comparison resolves into `true` or `false`, depending on the outcome. To get parts of a search pattern for further examination, such pattern recognition tasks can return groups of parts that have been found. There are also repetitions of groups to be reused within the expression to create more complex properties. Later, I'll show how this can be used. For now, I'll show several examples to

explain the creation of properties in a pattern.

Characters, Lines, and Text

I have already used terms such as *character* and *text*. Important in the context of regular expressions is the *term line*, too. Lines end with a line break (that's where you hit the Enter key in text processor). Many termination characters use line breaks. Files are often read line by line. That's why regular expressions handle lines explicitly. You need this knowledge to search beyond line endings.

Line Breaks

If you have text that doesn't contain any line breaks, you don't need the metacharacters for line begin and end. They simply do nothing.

Terms for Regular Expressions

Here I'm going to explain the more special terms.

Metacharacters

Within a regular expression, you can define special properties with metacharacters. Some of these you already know—particularly the `^` for the beginning and `$` for the end position in the search text.

`^` and `$` are called *metacharacters*. If you want to look for a metacharacter as a regular character, you have to put a backslash in front of it:

```
\$, \^
```

The backslash destroys the special meaning. Hence, the backslash itself is a metacharacter, too. If you look for the backslash in particular, you write `\\`.

A more thorough explanation of all metacharacters follows shortly .

Literals

Any language for programming has several literals to express things apart from keywords, such as numbers, operators, or text. JavaScript is no exception here. Regular expressions in JavaScript use `/` (forward slash) as a literal character.

```
/[abc]/
```

The expression can be assigned to a variable or used directly as a parameter.

```
1   var patt = /abc/;
2   var s = /abc/.toString();
3   console.log(s);
```

Character Classes

If you look for strings or single characters, it's often very efficient to use character classes. Character classes are written in square brackets, such as `[abc]`. The whole expression represents one (1) character out of the class. In the example, it's either `a`, or `b`, or `c`. It's neither `ab` nor `bc`. You must use a repetition operator to tell the expression engine to look for more appearances of the characters from such a class.

It's possible to write any character you're looking for one by one, or create groups by using the dash. The expression `[a-c]` would also hit the letter `b`.

References

Parts of successfully found text portions are stored in temporary memory cells. You can reference to them later in the same expression. The references use a backslash followed by the storage location . It looks like `\4`, for example.

Metacharacters Overview

The following is a complete list of the meta characters available in JavaScript 's regular expressions.

Start, End, and Boundaries

The most important ones you already know from the introduction chapter:

- `^` is the beginning (or start)
- `$` is the end
- `\b` defines the boundary of a word
- `\B` defines no boundary

What does that mean? If you look for the word “auto”, the “a” must be at the beginning, while “o” is the end. To search this string exactly (and nothing before and after), your expression is `^auto$`. If the word you’re looking for is anywhere in the text, the expression is simply `auto`.

```
1   var patt = /^auto$/;
2   console.log(patt.test("auto"));
3   console.log(patt.test("automatic"));
4
5   var patt2 = /auto/;
6   console.log(patt.test("That's our automobile,
it's an BMW."));
```

The output shows the result, as seen in Figure 2-1.

Figure 2-1. Output of the script

The end is mostly the end of a line. This might not be the case if some additional switches are set. Switches are outside the expression and control the regex engine. You can find more about them in Chapter 3.

Word boundaries are transitions from a word to surrounding parts. A word in Latin text ends with the appearance of a space or punctuation, such as a comma, semicolon, or period. The special symbol `\b` looks for such a transition. It does not exactly represent a character; it’s more something in between. That’s what I meant with the term *property* in the introduction. Regular expressions describe the properties of a search text, not the actual search term.

The expression `/\bco` recognizes “co” in the sentence “It is complicated.” The space before the “co…” creates the word boundary. The expression will find nothing in the sentence “Oncology is difficult.”—even if “co” is in the text as well.

Any Character

Often you'll look for a character at a particular position, but it doesn't matter which character. It's painful to handle the whole supply of characters in classes. That's why there is another metacharacter:

- `.` is any character, exactly

Now you can look for "auto" in different contexts:

- `.uto` looks for "auto", "Auto", "automatic", and so on, but also for "distributor". The latter is possibly not expected.

```
1   var patt = /.uto/;
2   console.log(patt.test("auto"));
3   console.log(patt.test("automatic"));
4   console.log(patt.test("distributor"));
```

The output shows the result, as seen in Figure 2-2.

Figure 2-2. Output of the script

No Characters

If you're not looking for anything, a regular expression doesn't seem useful. But it is. Imagine you're looking for empty lines in a file. That's what you'd find using just the end character:

```
^$
```

Yes, it's that easy, because the expression defines that the only characters are the line start and end. Hence, the line must be empty. And that's what you're looking for. Empty means here "nothing in between."

A single `^` as a search pattern does not make any sense. It just declares that there must be a beginning. That's a property any line possesses. Any text will match this pattern.

Character Classes

Character classes define groups of classes.

One out of Many

Character classes are written in square brackets. Without other operators, it's always just one position in the pattern. Groups are built by a dash and multiple groups are simply written concatenated. Any character in the class will be treated "as is" and loses its special meaning as a metacharacter.

- `[aeiou]` defines a vowel in English
- `[a-f]` defines the letters a, b, c, d, e, f (lowercase only)
- `[a-fA-F0-9]` defines all digits for hexadecimal numbers

The order is only important in groups. Simply put, the expressions `[a-fA-F]` and `[A-Fa-f]` are identical.

```
1   var patt = /[a-f]+/;
2   console.log(patt.test("Auto"));
3   console.log(patt.test("42"));
4   console.log(patt.test("12 Days"));
5   console.log(patt.test("borrow"));
```

The output shows the result, as seen in Figure 2-3.

Figure 2-3. Output of the script

Negations

A whole character class can be negated against the complete inventory of characters by using the metacharacter `^`. It has a different meaning and nothing in common with the start line metacharacter that uses the same sign. It's simply a different context. Some characters have a different meaning depending on the

context. That's why it is crucial to separate the parts of regular expressions carefully. You must understand the context of the character.

- `[^0-9]` includes everything but digits (that includes characters such as `#`, `*`, or `%`).
- `[^aeiou]` defines all consonants, but also a lot more, such as digits.

The special meaning of `^` works only if it is the very first character after the opening bracket. If it is somewhere else in the class, it's just a common character of its own:

- `[!"@§$%^&/()=]` defines all the characters on a keyboard's number keys in Shift level.

Digits

Seeking digits is a common task. Numbers are formed by digits and use a leading sign, a decimal point, and thousands mark. The following expressions might be helpful:

- `[0-7]` are digits of octal numbers (base 8)
- `[0-9+-.] are` decimal numbers with a sign and decimal point
- `[a-fA-F0-9]` are hexadecimal digits

Date and Time

The following expressions show how to recognize fragments of date and time within text:

- `[0-9/]` Date (American format, such as 4/22/2016).
- `[0-9:]` Time, such as 7:44
- `[0-9:amp]` Time with "am" or "pm"

The last example is often too simple, as it allows "a9" or "p0m".
The others are weak, because the time 26:99 would be found as well.

```
1   var patt = /[0-9:amp]/;
2   console.log(patt.test("12am"));
3   console.log(patt.test("4:17"));
```

The output shows the result, as seen in Figure 2-4.

Figure 2-4. Output of the script

> **Limitations**
> These examples show that simple regular expressions have clear limitations. While it is possible to write expressions that match the range of complex values perfectly, the effort can be huge. You can find better examples in the appendix.

Strings

By using character classes, you can easily distinguish between lowercase and uppercase letters:

- `[gG]reen, [rR]ed`

If a name might not be written in a distinct way you can use classes:

- `M[ae][iy]er` matches “Meyer”, “Mayer”, “Maier”, and “Meier”.

Abbreviations

Metacharacters are commonly formed as abbreviations, as listed and described in Table 2-1.

Table 2-1. Metacharacter Abbreviations

Abbreviation	Description
`\t`	Tabulator
`\n`	New line
`\r`	Return (carriage return)
`\f`	Form feed
`\v`	Vertical tabulator
`\s`	White space (not visible in print, includes `\t`, space, `\n`, `\r`, `\f`)

`S`	Negation of \s
`\w`	Word character (letters that form words, such as in `[_a-zA-Z0-9]`)
`W`	The negation of `\w`
`\d`	Digits, like `[0-9]`
`D`	Negation of `\d`
`\b`	Word boundary, start, or end of a word; all not in the \w definition.
`B`	Negation of `\b`
`\0`	Nul (`nil`) character (physical 0)
`\xxx`	Character value, written as an octal number
`\xdd`	Character value, written as an hexadecimal number
`\uxxxx`	Unicode character, written as hex number
`\cxxx`	Control, ASCII value

Repetition Operators

All metacharacters shown so far have one thing in common: they address exactly one character. If you need a number of occurrences, you use repetition operators. Some specials ones were mentioned in the introduction. Here I'll give a more complete description. But let's start with some simple examples:

- `a*` defines no or any occurrence of `"a"`, such as `"a"`, `"aaaa"`, and so on.
- `a+` defines one or any number of `"a"`, such as `"a"`, `"aa"`, and so on.
- `a?` defines no or one `"a"`, such as `""` or "a" exactly.

Common Operators

Common operators use a range within a number of characters. The following expressions use common operators:

- `{min, max}` Declares `min` characters that are required and `max` characters that are allowed
- `{wert}` The exact number
- `{,max}` The minimum can be omitted (zero to `max`)
- `{min,}` The maximum can be omitted as well; `min` to any number
- `{,}` Both ranges can be omitted (see the next paragraph for more about this construct)

Of course, you can always use this expression instead of the shorthand *, +, and ?. The verbose format would look like this:

- `{,}` means *
- `{0,1}` means ?
- `{1,}` means +

These operators are very good for text with a clearly defined length, such as zip codes:

```
<form>
  <input type="text" name="zip" value="">
  <input type="submit">
</form>
```

The following script checks whether the zip codes entered by the user make sense:

```
<script>
var re = /^[0-9]{5}$/;
var field = "12683";
var checkzip = re.exec(feld);
if(!checkzip) {
  alert("The zip code " + checkplz + " is not correct.");
} else {
    console .log(checkplz)
}
</script>
```

The output shows the result, as seen in Figure 2-5.

Figure 2-5. Output of the script

Summary

Table 2-2. Repetition Operators

Operator	Meaning	Description
`?`	0 – 1	None or one
`*`	0 – ∞	None or any
`+`	1 – ∞	One or any
`{num}`	Number	Exactly *num*
`{min,}`	Min – ∞	Minimum *min*
`{,max}`	0 – Max	None or a maximum of *max*
`{min, max}`	Min – Max	Minimal *min* and maximum of *max*

Special Operators

The “or” operator is a special operator written as `|`. For instance, `/green|red/` matches “green” in “green apple” and “red” in “red apple”, but it does not match “red or green apple”.

References

The elements shown so far look easy, right? More complex combinations of metacharacters and classes are not yet flexible enough. What’s really missing (when compared with traditional programming languages) are loops. When investigating strings in particular, a consecutive character-by-character search is quite helpful.

To get something similar in regular expressions, you can use references. That’s some sort of a special metacharacter, such as `\1`, `\2`, and so on. These numbered parts reference groups of characters found earlier in the search text. Groups that count are written in parentheses (…).

It’s sometimes hard to read the parentheses in the right way, because now you have all sorts of braces that can even appear nested. The best way is a simple pragmatic approach: just count the opening (left) parentheses. That’s what matters. The block of the first one is referenced by `\1`, the second by `\2`, and so on. JavaScript supports up to nine references. Other languages may support more.

Be careful

> Within character classes, you cannot use references. That's because the backslash has no special meaning herein. However, the part that is being formed by a character class can be put in parentheses and act as a reference.

The expression `/apple(,)\sorange\1/` will match "apple, orange," in the string "apple, orange, cherry, peach". The comma is bound in the first group and hence it's present after "orange" as well. Watch the `\s` in the middle, too. It's the space metacharacter.

Groups

The groups used for references can be used for repetition operators, too. Using groups for references is merely a side effect.

Simple Groups

If you want to repeat a group of characters, just use parentheses: `(ab)+` finds a match in `"abc"`, `"abcabc"`, and so on, but not in `"aacbb"`.

```
1   var patt = /(ab)+/;
2   console.log(patt.test("abc"));
3   console.log(patt.test("abcabc"));
4   console.log(patt.test("aacbb"));
```

The output shows the result, as seen in Figure 2-6.

Figure 2-6. Output of the script

Enclosing Characters

Expressions are a frequently used search pattern, where a text is enclosed by a pair of identical characters. That's often the literal appearance of strings in programming languages or the tags in HTML. Watch these text fragments:

- “Word Word” shall be found
- “Word Word’ shall not be found
- ‘Word Word’ shall be found

A possible solution for a regular expression could look like this:

```
/^(["']){1}.*\1$/
```

How does that work? The expression starts with `^` and then follows either a " (double quote) or a ' (single quote) `["']`. The whole part is enclosed in parentheses so it counts for a reference and shall appear once `{1}`. Then, any number of characters can follow `.*` until immediately before the end `$`, when the very same quote must appear. That’s written by `\1`. This clearly shows the nature of references.

They do not repeat the definition; they repeat the part being found by the references group.

```
1   var patt = /^(["']){1}.*\1$/;
2   console.log(patt.test("\"Word Word\""));
3   console.log(patt.test("\"Word Word\'"));
4   console.log(patt.test("\'Word Word\'"));
```

The output shows the result, as seen in Figure 2-7.

Figure 2-7. Output of the script

Because this very constructed example looks for quoted words by just repeating a word twice, it would be a nice exercise to handle exactly this. Instead of quotes, we’re looking for two identical words :

```
:/\b((\w+)\s+\2)+/
```

Because words have word boundaries, a `\b` starts the expression.

Then a group starts, and in it, another group that looks for word characters

(\w) that have to appear one or more times (+). That is followed by a one or more spaces (\s+). The whole group is repeated once again (\2). The repeated part is the inner group, because the 2 refers to the second opening parenthesis.

```
_ var patt = /\b((\w+)\s+\2)+/;
console.log(patt.test("Script Script istdoppelt"));
```

Non-Counting Groups

Sometimes it's hard to read an expression with many parentheses, from which only a few (or just one) are being used as a reference. For this, you can write (:…) to form a non-counting group.

Lookahead References

Looking back to a reference is only one part of the deal. Sometimes a match is valid, if another character follows. That can be achieved by lookahead references. Two metacharacter combinations are useful for this:

- ?= A positive lookahead; the following character must match
- ?! A negative lookahead; the following character must not match

```
1   var patt = /(\d{1,3})(?=d)/;
2   var text = "Duration: 16d";
3   var test = patt.exec(text);
4   console.log("Days: " + test[0]);
```

The output shows the result, as seen in Figure 2-8.

Figure 2-8. Output of the script

This example looks for numbers with one to three digits, which are immediately followed by the letter "d". The letter "d" itself is not part of the match—it just has to be there to make the digits match.

lookahead and lookbehind
You may imagine that if there is something looking ahead, it may also be looking behind. Regular expressions of several languages and platforms have this feature. However, JavaScript does not support lookbehind references.

Jörg Krause, Introducing Regular Expressions, DOI 10.1007/978-1-4842-2508-0_3

3. The JavaScript Functions

Jörg Krause[1]

(1) Berlin, Germany

JavaScript provides an object named `RegExp`. You can create an instance by using `new` or the literal `//`.

The RegExp Object

I'll start with an overview and follow it with several examples.

Methods

Apart from the `RegExp` object, you can use several built-in string functions in JavaScript to deal with regular expressions. The object knows these methods:

- `exec()`: Executes the test and returns the first hit.
- `test()`: Executes the test and returns either `true` or `false`.
- `toString()`: Returns the expression as a string.

The `exec` method returns an object that has the following content:

- An array with the groups (counting groups only):
  - The first capture is in index [0]
  - If there are more groups, these follow in [1] .. [n]
- A property named `index` contains the position of the first hit, its value is zero based.
- A property named `input` contains the searched text.

The `RegExp` object itself contains some more information. Of course, you find the definition of the expression, but also a way to iterate to further results (because `exec` returns just the first one).

Option *g*

If you expect multiple hits, you should use option `g`, which means "global" and continues searching after the first hit.

The following script contains a very simple expression that leads to multiple hits. The subsequent call of the very same expression runs through all hits by increasing an internal pointer. That forces the `do`-loop to iterate.

```
    1   var text = "Than Ann answers all questions
again.";
    2   var patt = /a/g;
    3   var match = patt.exec(text);
    4   console.log(match[0] + " found at " +
match.index);
    5
    6   match = patt.exec(text);
    7   do {
    8       match = patt.exec(text);
    9       if (!match) break;
   10       console.log(match[0] +
   11                   " found at " + match.index);
   12   } while(true);
```

The output shows the result, as seen in Figure 3-1.

Figure 3-1. Output of the script

Properties

Here are some more properties help keep the expression quite flexible:

- `constructor`: A function that creates the `RegExp` object.
- `global`: Checks for option `g` (Boolean), global.
- `ignoreCase`: Checks for option `i` (Boolean), case insensitive.
- `lastIndex`: Index of the last hit.
- `multiline`: Checks for option `m` (Boolean), multiline.
- `source`: The expression itself.

The following script shows the `lastIndex` property in action:

```
    1   var text = "Than Ann answers all questions
again.";
    2   var patt = /a/g;
    3   var match = patt.exec(text);
    4   console.log(match[0] + " found at " +
match.index);
    5
    6   match = patt.exec(text);
    7   do {
    8         match = patt.exec(text);
    9         if (!match) break;
   10         console.log(match[0] + " found at " +
match.index);
   11         console.log( "Further search at " +
patt.lastIndex);
   12   } while(true);
```

The output shows the result, as seen in Figure 3-2.

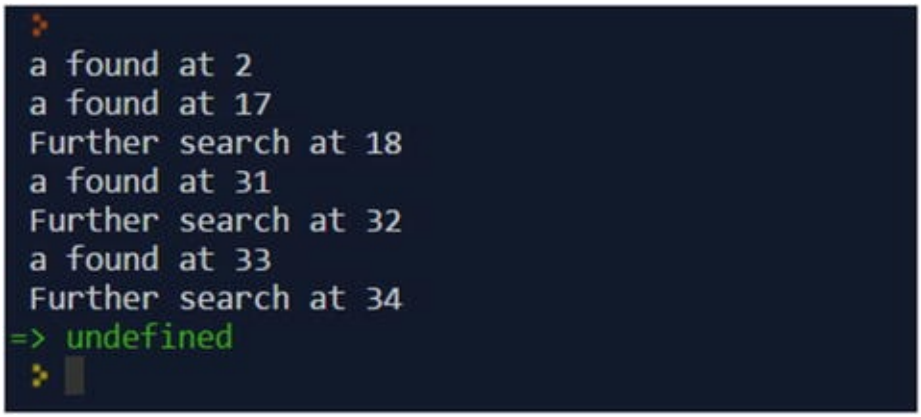

Figure 3-2. Output of the script

Dynamic Properties

If there are groups in the expression, they appear as dynamic properties. The names are `$1` to `$9`, respectively. If you don't want a group being catched this way, consider using a non-counting group `(?:)`.

```
1   var patt = /(abc)|(def)/;
2   var text = "anton def";
3   console.log(patt.test(text));
4   console.log(RegExp.$1);
```

The second output (line 4) returns the content of first match. That's `def`, actually.

The Literal Form

The literal form is the easiest way to create an expression:

```
1   var patt = /web/i;
2   patt.test("Look for our Web courses!");
```

This simply returns `true`. Without the `i` at the end, the result is `false`. The `i` suppresses case sensitivity and makes "Web" match "web" and vice versa. You can even combine this with `exec`:

```
1   /web/i.exec("Look for our Web courses!");
```

This will return an object with the hit "Web" at first position and the value 13

for the `index`. If there is no hit, `exec` simply returns `null`. Use `if(//.exec(""))` to handle this, because JavaScript will treat `null` as `false`. Using the `new` operator, it would look like this:

```
1   var patt = new RegExp("web");
```

Literals and Regex

While `//` is valid for RegExp, a regular string will work here as well—but only here.

Execution Options

Some options are already shown. The options are not part of the expression; instead, they modify the behavior of the regex engine.

```
var pattern = /[A-Z]/i;
```

These are all the available options:

- `i`: Searches case insensitive
- `g`: Searches globally, even if a hit was already found
- `m`: Searches over multiple lines (without this, a line end stops the search)

String Functions

Regular expressions are not limited to the `RegExp` object. Some regular string functions can handle these expressions, too.

Overview

The following functions are enhanced to handle expressions:

- `search`
- `replace`
- `match`
- `split`

```
1   var str = "Look for our Web courses!";
2   var res = str.search(/Web/i);
3   console.log(res);
```

The `res` variable contains 13. That's the index of the hit. Without a hit, you get –1.

If you need a Boolean value, just use `match`:

```
1   var str = "Look for our Web courses!";
2   var res = str.match(/Web/);
3   if (res) {
4     console.log("Hit");
5   } else {
6     console.log("No Hit");
7   }
```

Here you get *Hit* on the console. It's quite similar using `replace`:

```
1   var str = "Look for our Web courses!";
2   var res = str.replace(/Web/i, ".NET");
3   console.log(res);
```

The `res` variable now contains *Look for our .NET courses!*.

Jörg Krause, Introducing Regular Expressions, DOI 10.1007/978-1-4842-2508-0_4

4. Examples of Patterns

Jörg Krause[1]

(1) Berlin, Germany

You have to do some tasks again and again in your life as a developer. This chapter contains several useful examples.

Web and Network

The expressions in this section are dedicated to web and network environments.

HTML Tags

This is how you look for an HTML tag with the fixed name `DIV`:

```
/<DIV\b[^>]*>(.*?)<\/DIV>/i
```

And this expression looks for any tag:

```
/<([A-Z][A-Z0-9]*)\b[^>]*>(.*?)<\/\1>/i
```

Once again, I'm using a `\1` reference on the first hit to find the matching closing tag.

```
1   var patt = /<([A-Z][A-Z0-9]*)\b[^>]*>(.*?)
<\/\1>/i;
2   var html = "Ww want write <b>bold</b> and
<i>italic</i> here";
3   console.log(patt.exec(html));
```

The output shows the result, as seen in Figure 4-1.

Figure 4-1. Output of the script

Figure 4-1 shows only the first hit. Execute again to get further hits .

IP Addresses

With IP addresses, you can see how you get the regex engine's limitations. Search patterns are not that good with numbers. With some real effort, it's still possible, but conventional programming might be a better option. Let's start with a naive approach first:

```
\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b
```

That works, but also does for 999.999.999.999. And that's definitely not a valid IP. But how do you limit the range to 0–255? Because the four parts are identical, it's all about creating a pattern for just one part and then use references. The trick needed is to stop thinking in numbers and treat the digits separately. The unit goes from 0 to 9, but if the hundreds are greater than 250, it only goes from 0 to 5. For the tens, it's similar: 0 to 9, or for numbers from 200, it's 0 to 5. So we can get a range of numbers from 0 to exactly 255. And here you go:

```
/\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
  (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
  (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
  (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b/
```

The expression reveals the segments as single groups. If the groups are not required, the whole expression can be shortened:

```
/\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)
```

```
{3}
     (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b/
```

That wasn't bad, but it probably looks somewhat scary. Again, dealing with number ranges is not the strength of regular expressions.

Mac Addresses

MAC addresses are hardware-related identifiers that usually identify network endpoints. It's a 48-bit number that manufacturers burn into the hardware.

 Some hardware allows this number to be overwritten by software or settings, so it can be spoofed to be honest. If you use regular expressions to identify such numbers, keep this in mind.

The general form is a series of two hexadecimal digits, divided by colons, spaces, dashes, or even nothing. In a pretty form, it looks like this: CB:35:2F:00:7C:A1. Here is the expression:

```
/^[0-9A-F]{2}([-: ]?[0-9A-F]{2}){5}$/i
```

The following expression checks for the same, but returns all parts as groups for further investigation:

```
/^([\dA-F]{2})[-: ]?
   ([\dA-F]{2})[-: ]?
   ([\dA-F]{2})[-: ]?
   ([\dA-F]{2})[-: ]?
   ([\dA-F]{2})[-: ]?
   ([\dA-F]{2})$/i
```

To get access to the groups' content, you'll need a loop in JavaScript:

```
1   var patt = /^([\dA-F]{2})[-: ]?([\dA-F]{2})[-:
]?([\dA-F]{2})[-: ]?(\
2    [\dA-F]{2})[-: ]?([\dA-F]{2})[-: ]?([\dA-F]
{2})$/i;
3   var mac = "CB:35:2F:00:7C:A1";
4   mac = patt.exec(mac)
```

```
5    for(i = 1; i < mac.length; i++) {
6      console.log(mac[i]);
7    }
```

The output shows the result, as seen in Figure 4-2.

Figure 4-2. Output of the script

URL

The next expression checks a string for the common URL (Unified Resource Locator) format. The URL starts with the protocol (sometimes called *schema*) and here we allow http, https, and ftp, respectively. Casing doesn't matter. The divider between protocol and the data is `://`. The first attempt is simple, but it will accept a few invalid forms and it rejects special forms of valid ones. The main disadvantage is that the URL can contain user credentials in the format (*`https://joergs:password@joergkrause.de/regex/`*). Anyway, sometimes it's quite enough:

```
/^([Hh][Tt][Tt][Pp][Ss]?|[Ff][Tt][Pp]):\/\/(.+)$/
```

A more comprehensive form would look like this (embedded in a script):

```
1    var patt = /^((([hH][tT][tT][pP][sS]?|[fF][tT]
[pP])\:\/\/)?([\w\.\-]\
2    +(\:[\w\.\&%\$\-]+)*@)?(([^\s\(\)\<\>\\\"\.\
[\]\,@;:]+)(\.[^\s\(\)\\
3    <\>\\\"\.\[\]\,@;:]+)*(\.[a-zA-Z]{2,4}))|
((([01]?\d{1,2}|2[0-4]\d|25\
4    [0-5])\.){3}([01]?\d{1,2}|2[0-4]\d|25[0-5])))
(\b\:(6553[0-5]|655[0-2\
5    ]\d|65[0-4]\d{2}|6[0-4]\d{3}|[1-5]\d{4}|[1-
9]\d{0,3}|0)\b)?((\/[^\/]\
```

```
   6   [\w\.\,\?\'\\\/\+&%\$#\=~_\-@]*)*[^\.\,\?\"\'\
(\)\[\]!;<>{}\s\x7F-\x\
   7   FF])?)$/;
   8   var url =
"http://www.joergkrause.de:8080/index.php?id=0";
   9   console.log(patt.test(url));
```

That’s definitely worth further investigation. Let’s split the expression into the relevant parts:

```
    1   /^
      // Start
    2     (
      //
    3       (
      //
    4         (
      //
    5           [hH][tT][tT][pP]
[sS]?                        // http
    6           |
      // or
    7           [fF][tT]
[pP]                                  // ftp
    8         )
      //
    9         \:\/\/
      // :// token
   10       )?
   11       (
   12         [\w\.\-
]+                                       // Domain
   13         (\:[\w\.\&%\$\-
]+)*@                             //
   14       )?
   15       (
   16         (
   17           ([^\s\(\)\<\>\\\"\.\
[\]\,@;:]+)             // Parts of domain*
```

```
  18            (\.[^\s\(\)\<\>\\\"\.\
[\]\,@;:]+)*          //
  19            (\.[a-zA-Z]{2,4}) // Top level domain
  20          )
  21          |
  22          (
  23            (
  24              ([01]?\d{1,2}|2[0-4]\d|25[0-5])\.)
{3}    // IP address
  25              ([01]?\d{1,2}|2[0-4]\d|25[0-
5])           //
  26            )
  27          )
  28          (\b\:
     // Port number:
  29            (6553[0-
5]                                  // upper part
  30             | //
  31             655[0-
2]\d                                 // ten thousends
  32             | //
  33             65[0-
4]\d{2}                               // thousends
  34             | //
  35             6[0-
4]\d{3}                                // Hundreds
  36             | //
  37             [1-
5]\d{4}                                 // Tens
  38             | //
  39             [1-
9]\d{0,3}|0)\b                          // Ones
  40           )?
     //
  41            (
     // path or file
  42             (\/[^\/][\w\.\,\?\'\\\/\+&%\$#\=~_\-
@]*)*
  43              [^\.\,\?\"\'\(\)\[\]!;<>{}\s\x7F-
```

```
\xFF]
   44            )?
   45        )$/
```

Actually, the domain defines the characters that are not allowed, rather than the ones that are allowed. The top level is limited to two to four characters, which excludes some newer domains. So in reality, this expression may need further tweaking.

Query String

The part after the URL is called the *query string*. The divider is the question mark `?`. This query string usually has a specific format, which is what the following expression is looking for:

```
/([^?=&]+)(=([^&]*))?/g
```

This is another example of the limits of regular expressions. You need some code to make it work properly. Doing it completely in regular expressions would be become extremely complex. Sometimes, it's better to use a simple expression in conjunction with simple code to get things done, rather than using a huge expression or some complex code on its own.

```
    1   var uri = 'http://joergkrause.de/index.php?
cat=113&prod=2605&query=a\
    2   press';
    3   var queryString = {};
    4   uri.replace(
    5       /([^?=&]+)(=([^&]*))?/g,
    6       function($0, $1, $2, $3) { queryString[$1]
= $3; }
    7   );
    8   for (var i in queryString){
    9       if (!queryString[i]) continue;
   10     console.log(i + " = " + queryString[i]);
   11   }
```

This code assumes that the query string has a chain of key=value pairs. The output shows the result, as seen in Figure 4-3.

Figure 4-3. Output of the script

Port Numbers

Port numbers go from 1 to 65535. This is once again a number-check exercise, which I'm going to explain further later in this chapter. But first I'll show the final expression:

```
^(4915[0-1]
 |491[0-4]\d
 |490\d\d
 |4[0-8]\d{3}
 |[1-3]\d{4}
 |[2-9]\d{3}
 |1[1-9]\d{2}
 |10[3-9]\d
 |102[4-9])$
```

In JavaScript, it could be used like this (`\d` is the metacharacter for a digit):

```
 1   var patt = /^(4915[0-1]
 2               |491[0-4]\d
 3               |490\d\d
 4               |4[0-8]\d{3}
 5               |[1-3]\d{4}
 6               |[2-9]\d{3}
 7               |1[1-9]\d{2}
 8               |10[3-9]\d
 9               |102[4-9])$/;
10   var port = 1384;
11   console.log(patt.test(port));
12   port = 75000;
13   console.log(patt.test(port));
```

The output shows `true` (1384 is valid) and then `false` (75000 is not valid)
.

Manipulating Data

JavaScript's `replace` function supports regular expressions directly. That's the common way to change data based on search patterns.

Remove Spaces

Spaces can be removed quite easily. Try searching for `^[ \t]+` and replace the spaces with nothing. If you look for the spaces at the end, use this: `[ \t]+$`. If you want to remove spaces at the beginning and the end, use this: `^[ \t]+|[ \t]+$`. Instead of declaring all space chars individually, such as in `[ \t]` (space and tab only), you can even add more, such as line breaks: `[ \t\r\n]`. The latter can be abbreviated with the space metacharacter (`\s`).

```
   1   var patt = /[ \t]+/g;
   2   var text =
"Here•we•have•many••••Spaces\tand•Tabs•too";
   3   text = text.replace(patt, "");
   4   console.log(text);
```

The • character in the listing is just to make the spaces visible in print. The output shows the result, as seen in Figure 4-4.

Figure 4-4. Output of the script

Simulation of a Variable Distance

If the distance between the parts that you're looking for is a variable, the expression is a bit more challenging. Imagine that you're looking for two words close together, but the actual distance is not a constant value.

The pattern needs a definition for the first word and for the second, and then something for the space in between. For the undefined part of a word, the word

border `\w+` is helpful. The other parts use the opposite `\W+` metacharacter (no word character). `\b` is the border itself and claims no character.

The no expression looks like this:

```
\bthere\W+(?:\w+\W+){1,5}?here\b
```

The `{1,5}?` quantifier defines the tolerance of the distance. So this expression recognizes two identical words within a distance of one to five other words. Have a look at the text itself:

```
here and there is like there and here
```

The expression will find the words “there” and “here” (two words distance) in the middle of the sentence.

If you look for the opposite way, as well an OR operator, `|` is sufficient:

```
\b(?:there\W+(?:\w+\W+){1,6}?here |here\W+
(?:\w+\W+){1,6}?there)\b
```

If you want pairs of words, this could be written even more simply:

```
\b(word1|word2|word3) (?:\W+\w+){1,6}? \W+
(word1|word2|word3)\b
```

If you look for exact matches of the same word, you can simplify the expression further by using references:

```
1   var patt = /\b(there)\W+(?:\w+\W+){1,5}?\1\b/;
2   var text = "here and there is like there and
here";
3   console.log(patt.test(text));
```

This results in `true`. If you use “here” instead, you’ll get `false` (“here” is six words apart).

File Extensions

The following expression looks for file extensions:

```
/^.*(([^\.][\.][wW][mM][aA])|([^\.][\.][mM][pP]
[3]))$/
```

The actual example looks for `*.wma` and `*.mp3`. That's helpful for file upload scenarios.

Non-Printable Characters

This expression looks for non-printable characters:

```
/[\x00-\x1F\x7F]/
```

Hexadecimal Digits for Colors

Digits for color codes in CSS and HTML use the hexadecimal form (#FFFFFF or #333):

```
/^#(\d{6})|^#([A-F]{6})|^#([A-F]|[0-9]){6}/
```

Form Validation

Forms typically contain fields for URL, email, dates, or phone numbers. Wherever the form comes from, regular expressions can further refine the checks.

eMail

In this book, I started with an example for checking an email address. There are zillions of variations that you can find. Let's start with a simple one, which I'll refine further:

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
```

This is short, pretty, and—wrong. It does not recognize perfectly valid addresses. It even let some invalid ones slip through. Whatever you do for improvement, you first need your requirements. A regular expression that handles all valid forms of an email actually doesn't exist. But, 99.9% is often good enough, isn't it?

The first special part of the pattern is the usage of word boundaries (`\b`), which is the basis to find addresses within larger texts. If it is not a requirement, you can replace the boundaries with `^` and `$`, respectively.

Using functions and options

Removing spaces has been discussed already. If you can use functions such as `trim()`, it's often the better option. Dealing with letter cases is best handled with the `//i` option, rather than extended character classes like `[A-Za-z]`.

A critical part of email addresses is the top-level domain. As long as you want to only get `.com` or `.net`, life is easy. But newer top-level domains include extensions such as `.museum` and `.berlin`. The list is huge, and new top-level domain names become available all the time. So either you regularly extend your expression or you accept everything (that means *joerg@krause.g e eknet* would be perfectly valid).

You can, of course, limit the number of characters. Use `{2,4}` or extend it to `{2,6}`. The more letters you allow, the more valid top-level domains that you can catch—and a lot more invalid ones, too. Using only the top levels from the early stages of the Internet, this looks easy:

```
^[A-Z0-9._%+-]+@[A-Z0-9.-]+\. ' '
(?:[A-Z]{2}|com|org|net|edu|gov|mil)$
```

The country names are handled by `[A-Z]{1,2}`, which is usually acceptable. As long as you don't care about "xx" or "yy", the extension is easy:

```
1   `^[A-Z0-9._%+-]+@[A-Z0-9.-]+
2   \.(?:[A-Z]{2}|com|org|net|edu|gov|mil|biz
3   |info|mobi|name|aero|asia|jobs|museum)$`
```

A little more work happens if you add subdomains, such as in *joerg@server.firma.provider.com*. Because the simple form accepts dots, it would also allow this email address to pass. Unfortunately, the address *joerg@firma….de* is accepted as valid as well. The following expression prevents this:

```
\b[A-Z0-9._%+-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,4}\b
```

Since the dot becomes part of the character definition, each dot must have at least one character before it.

So things are going to get complex. To make it right, taking a look at the

Internet standard's definition is a good idea. For email, it's the RFC 5322. From this document, I have extracted the following syntax definition:

```
    1      (?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+
    2       (?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
    3       | "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-
\x5b\x5d-\x7f]
    4          | \\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
    5    @ (?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
[a-z0-9]
    6       (?:[a-z0-9-]*[a-z0-9])?
    7       |  \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9]
[0-9]?)\.){3}
    8            (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?
    9               |[a-z0-9-]*[a-z0-9]:
   10               (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-
\x5a\x53-\x7f]
   11               |  \\[\x01-\x09\x0b\x0c\x0e-
\x7f])+)
   12          \])
```

The `@` sign purposefully splits the expression into two parts. It also includes checking for an IP address. That's rarely used but perfectly valid. For the name, another odd part is acceptable. The `"` and `\` characters are valid if there is a backslash beforehand. This is extremely unusual , but RFC 5322 says it's allowed.

While it seems rather complicated, the expression in itself is weak. Again, it handles valid top-level domains, not as they are defined, but more globally. If we reduce the requirements a bit and disallow IP addresses and odd characters in names, it becomes a lot easier to read:

```
   1    [a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-
9!#$%&'*+/=?^_`{|}~-]+)*
   2    @
   3    (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9]
(?:[a-z0-9-]*[a-z0-9])?
```

Now I'm going to combine this with the part shown earlier:

```
1    [a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-
9!#$%&'*+/=?^_`{|}~-]+)*
2    @
3    (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
4      (?:[A-Z]{2}|com|org|net|edu|gov|mil
5                  |biz|info|mobi|name|aero
6                  |asia|jobs|museum|berlin)\b
```

Watch Top-Level Domains!

Again, watch for the list of currently valid top-level domains. A complete list can be found on the IANA website.[1] As you can see, `.bananarepublic` is a valid top-level domain , and so is `.wtf`. Right, WTF.

Date Expressions

For American date formats, I'll start with a naive form:

```
^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12]
[0-9]|3[01])$
```

The dividers are variable (2015-07-13 or 2015/07/13). They can be mixed, as well (2015/07-13). The mixed issue can be solved with a reference:

```
^(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|[12]
[0-9]|3[01])$
```

For the German form of a date, another expression is required:

```
(0[1-9]|[12][0-9]|3[01])[.](0[1-9]|1[012])[.]
(19|20)\d\d
```

That's the hassle—for each form, you need another expression. Consider combining your expression with the | (OR) operator.

The years are limited from 1900 to 2099. The month can be 01 to 09 or 10, 11, 12. Days are in the tens and go from 01 to 31. The divider is fixed to `.`, but you can easily change that.

It isn't perfect, however, because this expression allows 31.02.2016. Yet, we all know that the February never has 31 days, which is also the case for some of

the other months. Of course, you can handle it month by month. But checking for leap years is a real challenge in regular expressions. In a programming language it's easy: check whether you can divide the year by 4 but not by 100. That's it.

If a pattern has to be recognized, regular expressions are easy and powerful. If some logic comes into play, things get hairy. So let's use a combination of regular expression and code:

```
 1   function check_form(date)
 2   {
 3      var pattern = /(0[1-9]|[12][0-9]|3[01])[.]
(0[1-9]|1[012])[.](19|2\
 4   0)\d\d/;
 5      if(date.match(pattern))
 6      {
 7         var date_array = date.split('.');
 8         var day = date_array[0];
 9         // Monate sind intern 0-11
10         var month = date_array[1] - 1;
11         var year = date_array[2];
12         // Prüfung an JavaScript übergeben
13         source = new Date(year,month,day);
14         if(year != source.getFullYear())
15         {
16            console.log('Year wrong!');
17            return false;
18         }
19
20         if(month != source.getMonth())
21         {
22            console.log('Month wrong!');
23            return false;
24         }
25
26         if(day != source.getDate())
27         {
28            console.log('Day wrong!');
29            return false;
```

```
30          }
31        }
32        else
33        {
34            alert('Pattern wrong!');
35            return false;
36        }
37
38        return true;
39    }
```

So the expression is doing the rough part and the code needs further refinement.

If you're thrilled by regular expressions and you want to shine in front of your colleagues, you can at least handle the months like this :

```
1    ^(d{0}|
2       (31(?!(FEB|APR|JUN|SEP|NOV)))
3        |((30|29)(?!FEB))
4         |(29(?=FEB(((1[6-9]|[2-9]\d)(0[48]|[2468]
[048]|[13579][26])
5                  |((16|[2468][048]|[3579]
[26])00)))))
6        |(29(?=FEB(((0[48]|[2468][048]|[13579]
[26])
7                  |((16|[2468][048]|[3579]
[26])00)))))
8        |(0?[1-9])|1\d|2[0-8])
9         (JAN|FEB|MAR|MAY|APR|JUL|JUN|AUG|OCT|SEP
|NOV|DEC)
10      ((1[6-9]|[2-9]\d)\d{2}|\d{2}|d{0})$
```

This is a very nifty example of the usage of back references. There is *no* character between the day and the month, which means it also recognizes strings such as "23MAR2017".

Use Functions

You should always consider using functions if they drastically simplify your

expressions.

String Passwords

The following expression requires a password with these properties:

- 1 lowercase letter
- 1 uppercase letter
- 1 digit
- 1 special char
- At least 6 and max 50 characters long

```
/((?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[\W_]).
{6,50})/i
```

ISBN

The ISBN (International Standard Book Number) is a unique identifier for books. These numbers look like ISBN 9-783738-62519-6 or 9-783738-62519-6.

```
/^(ISBN )?\d-\d{3,6}-\d{3,6}-\d$/
```

The trailing number is a check value, which you can't handle with regular expressions. Using code is a lot easier here.

Currencies

With currency, there is a combination of digits and a currency symbol. The $ (dollar) character is a particular challenge, because it's also a metacharacter:

```
^\$[+-]?([0-9]+|[0-9]{1,3}(,[0-9]{3})*)(\.[0-9]
{1,2})?$
```

Number Ranges

Number ranges are always challenging. Assume that you want to check for 1 to 248. Something like [1-248] doesn't work. You must create a chain and check digit by digit.

The digits use either [0-9] or the `\d` metacharacter. For 1 to 99, you'd do

this:

```
1?[1-9]|[1-9][0-9]
```

1 to 199 is easier: `1[0-9][0-9]`. And `1\d{2}` works, too. Now let's handle a range from 200 to 248. The numbers higher than 40 need special treatment. First, let's look at 200 to 239:

```
2[0-3][0-9]
```

And now 240 to 248:

```
24[0-8]
```

Altogether, it looks like this:

```
1[1-9][0-9]|2[0-3][0-9]|24[0-8]
```

Here are some commonly used ranges:

- 000..255: `^([01][0-9][0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..255: `^([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..127: `^(0?[0-9]?[0-9]|1[01][0-9]|12[0-7])$`
- 0..999: `^([0-9]|[1-9][0-9]|[1-9][0-9][0-9])$`
- 000..999: `^[0-9]{3}$`
- 0 or 000..999: `^[0-9]{1,3}$`
- 1..999: `^([1-9]|[1-9][0-9]|[1-9][0-9][0-9])$`
- 001..999: `^(00[1-9]|0[1-9][0-9]|[1-9][0-9][0-9])$`
- 1 or 001..999: `^(0{0,2}[1-9]|0?[1-9][0-9]|[1-9][0-9][0-9])$`
- 0 or 00..59: `^[0-5]?[0-9]$`
- 0 or 000..366: `^(0?[0-9]?[0-9]|[1-2][0-9][0-9]|3[0-5][0-9]| 36 [0-6])$`

Floating Point Numbers

Floats are as challenging as integers. You need to handle the thousands divider (a dot (.) in Germany and a comma (,) in the United States) and the decimal point (a comma (,) in Germany and a dot (.) in the United States). Let's start with a simple version (US format):

```
[-+]?[0-9]*[.]?[0-9]+
```

Using the `*` makes everything before the decimal point optional. That would allow "–.6", which is probably fine, although odd. This is a better example:

```
'[-+]?([0-9]*[.][0-9]+|[0-9]+)
```

Writing a power, looks like this:

```
^(-?[1-9](.\d+)?)((\s?[X*]\s?10[E^]([+-]?\d+))|
(E([+-]?\d+)))$
```

This matches 1.7e5 or 22e^10.

Thousands Divider

The thousands divider uses a comma (US format):

```
(?:^(?:-)?(?:\d{1,3},(?:\d{3},)*\d{3})(?:\.\d+)?$|
^(?:-)?\d*(?:\.\d+)?$)
```

Credit Cards

Credit cards have a very regular structure and are handled well by regular expressions.

- **Visa:** `^4[0-9]{12}(?:[0-9]{3})?$` All Visa numbers start with "4 and have 16 digits.
- **MasterCard:** `^5[1-5][0-9]{14}$` All MasterCard numbers start with 51 to 55 and are 16 digits.
- **American Express:** `^3[47][0-9]{13}$` American Express starts with 34 or 37 (Gold) and have 15 digits.
- **Diners Club:** `^3(?:0[0-5]|[68][0-9])[0-9]{11}$` Diners Club

starts with 300 to 305 or 36 or 38. They use 14 digits.

The following expressions make the deal:

```
1   ^(?:4[0-9]{12}(?:[0-9]{3})?          # Visa
2     | 5[1-5][0-9]{14}                  # MasterCard
3     | 3[47][0-9]{13}                   # American Express
4     | 3(?:0[0-5]|[68][0-9])[0-9]{11}   # Diners Club
5   )$
```

Cards have also have check digits that use the *Luhn algorithm*. It definitely requires a piece of code:

```
1   function valid_credit_card(value) {
2     // Add the former expression here
3     if (/[^0-9-\s]+/.test(value)) return false;
4
5         // The LUHn algorithm
6         var nCheck = 0, nDigit = 0, bEven = false;
7         value = value.replace(/\D/g, "");
8
9         for (var n = value.length - 1; n >= 0; n--) {
10                var cDigit = value.charAt(n),
11                        nDigit = parseInt(cDigit, 10);
12
13                if (bEven) {
14                        if ((nDigit *= 2) > 9) nDigit -= 9;
15                }
16
17                nCheck += nDigit;
18                bEven = !bEven;
19        }
```

```
   20
   21          return (nCheck % 10) == 0;
   22    }
   23
   24    console.log(valid_credit_card('401288888888188
1'));
```

You can use the previous expression shown to replace the `\d+`. The expression accepts only the numbers—not the spaces or dashes added to improve readability. Strip these characters before you start your check procedure.

Test Data

To check your code, you need credit card numbers. It's always a very bad idea to use actual credit card numbers, however. That's why the credit card issuers have test numbers that will be never possessed by anyone.

Table 4-1. Test Ranges for Credit Cards

Credit Card Type	Credit Card Number
American Express	378282246310005
American Express	371449635398431
AmEx Corporate	378734493671000
Diners Club	30569309025904
Diners Club	38520000023237
MasterCard	5555555555554444
MasterCard	5105105105105100
Visa	4111111111111111
Visa	4012888888881881
Visa	4222222222222

Geo Coordinates

Once again, let's start with a simple example:

```
   ^[0-9]{1,2}°[0-9]{1,2}[0-9]{1,2} [NnSs] [0-9]
{1,2}°[0-9]{1,2}[0-9]{1,2} [WwEe]$
```

This handles the traditional forms, such as in 52°31′27″ N 13°24′37″ E.

For other schemas, it could look like this:

```
   1   var ck_lat = /^(-?[1-8]?\d(?:\.\d{1,18})?
   2                |90(?:\.0{1,18})?)[EW]?$/i;
   3   var ck_lon = /^(-?(?:1[0-7]|[1-9])?
\d(?:\.\d{1,18})?
   4                |180(?:\.0{1,18})?)[NS]?$/i;
   5
   6   function check_lat_lon(lat, lon){
   7     var validLat = ck_lat.test(lat);
   8     var validLon = ck_lon.test(lon);
   9     return (validLat && validLon);
  10   }
  11
  12   console.log(check_lat_lon("13E", "52N"));
```

The scripts uses the `i` operator for the cardinal points.

Guid/UUID

A guid is a *global unique identifier*, a 2^128 number, written in 32 hex digits. A typical pattern can look like this:

- [{hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh}]
- [hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh]
- [hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh]
- [0xhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh]

The letter `h` means *hex digit* (0-F). This is how you check for it:

```
   1   /^((?-i:0x)?[A-F0-9]{32}|
   2                [A-F0-9]{8}-
   3                [A-F0-9]{4}-
   4                [A-F0-9]{4}-
   5                [A-F0-9]{4}-
   6                [A-F0-9]{12}|
   7              \{[A-F0-9]{8}-
   8                [A-F0-9]{4}-
   9                [A-F0-9]{4}-
```

```
10                  [A-F0-9]{4}-
11                  [A-F0-9]{12}\})$/i
```

Again, the `i` option is used because you can write hex with an “a” as well as with “A”.

Percentages

Here we start with a simple pattern for US format numbers:

```
(?!^0*$)(?!^0*\.0*$)^\d{1,2}(\.\d{1,4})?$
```

In German, you would use this expression:

```
(?!^0*$)(?!^0*,0*$)^\d{1,2}(,\d{1,4})?$
```

Footnotes

1 http://data.iana.org/TLD/tlds-alpha-by-domain.txt

Appendix

As a quick reference, here you will find all the metacharacters and their descriptions.

Table A-1. Characters

Abbreviation	Description
`.`	Any character
`[]`	One out of an inventory of characters
`[^]`	One not in the inventory of characters
`.`	The dot
`\`	Masking special chars
`\\`	The backslash

Table A-2. Groups

Abbreviation	Description
`()`	Counting group.
`(?:)`	Non-counting groups.
`(?=)`	Lookahead reference. Matches if the next character matches.
`(?=!)`	Negative lookahead reference. Matches if the next character does not match.

Table A-3. Operators

Operator	Meaning	Description
`?`	0 – 1	Zero or one character(s)
`*`	0 – ∞	Zero or any number of characters
`+`	1 – ∞	One or any number of characters
`{zahl}`	number	Exact “number” of characters
`{min,}`	min – ∞	Minimum number of characters
`{,max}`	0 – Max	No or maximum of number of characters
`{min, max}`	Min – Max	Minimum up to maximum characters
`^`		Start; with option “m” the beginning of a line
`$`		End; with option “m” the end of a line
`|`		Logical OR

Table A-4. Abbreviations

Abbreviation	Description
`\t`	Tabulator character
`\n`	Newline
`\r`	Return (carriage return)

\f	Form feed (page break)
\v	Vertical tabulator
\s	White space (non-printable character, such as \t, space, \n, \r, \f)
S	Negation of \s
\w	Word character (character from that one build words, especially [_a–zA–Z0-9]
W	The negation of \w
\d	Digit, same as [0–9]
D	Negation of \d
\b	Word boundary, start and end of word found by recognizing all chars not part of \w.
B	Negation of \b
\0	Null character (physical 0)
\xxx	Character value, represented as octal number
\xdd	Character value, represented as hexadecimal number
\uxxxx	Unicode character in hexadecimal form
\cxxx	Control character, ASCII value

Table A-5. JavaScript Functions

Name	Description
exec	RegExp method , checks and returns an array
test	RegExp method, checks and returns a Boolean
match	String method , array or null
search	String method, index of first hit or –1
replace	String method, replaced string or same if no hit
split	String method, array
//o	Literal of the RegExp objects (o=Option, see Table A-7)

Table A-6. Options

Abbreviation	Description
g	Global, continue after first hit
m	Multiline, treat line breaks as regular characters
i	Ignore case sensitivity

Index

A, B

Abbreviation

C

D

E

F

G

H

I

J, K

L

M

N

O

P

Q

R

S, T

U, V

W, X, Y, Z