Regular expressions are patterns or templates that allow you to define a set of rules in a natural yet vague way, giving you the ability to match and validate text. Therefore, they have been implemented in nearly every modern programming language. JavaScript’s implementation allows us to perform complex tasks with a few lines of code using regular expressions to match and extract data out of text.

This book starts by exploring what a pattern actually is and how regular expressions express these patterns to match and manipulate user data. You then move on to learning about the use of character classes to define a wild character match, a digit match, and an alphanumeric match. You will then learn to manipulate text and shorten data in URLs, paths, markup, and data exchange, as well as other advanced Regex features.

Finally, you will work through real-world examples, both in the browser and on the server side using Node.js.

Who this book is written for
This book is ideal for JavaScript developers and programmers who work with any type of user entry data and want sharpen their skills to become experts.

What you will learn from this book
- Structure your patterns and model different types of constraints
- Clean and optimize code with Regex’s processing power
- Solve common use cases and situations
- Extract different types of fields correctly
- Embed regular expressions both in your UI and on the backend
- Efficiently create guard clauses in your functions using patterns
- Create blank capture groups to handle inconsistent data
- Capture parts of a pattern in a group to display, replace, or use character sets to match characters

Leverage the power of regular expressions to create an engaging user experience
In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'Getting Started with Regex'
- A synopsis of the book’s content
- More information on **JavaScript Regular Expressions**

### About the Authors

**Loiane Groner** has over 9 years of software development experience. In her university, she demonstrated a great deal of interest in IT. She worked as a teacher's assistant for 2.5 years for algorithms, data structures, and computing theory. She represented her university at the ACM International Collegiate Programming Contest – Brazilian Finals (South America Regionals), and she also worked as the student delegate of the SBC (Brazilian Computing Society) for 2 years. Loiane won a merit award in her senior year for being one of the top three students in her course. She had one of the highest GPAs in the computer science department, and also graduated with honors.

She has already worked at multinational companies, such as IBM. Her areas of expertise include Java SE and Java EE and also Sencha technologies (such as Ext JS and Sencha Touch). Nowadays, Loiane is working as a software development manager at a financial institution where she manages overseas solutions. She also works as an independent Sencha consultant and coach.


She is passionate about Sencha and Java, and she is the CampinasJUG (Campinas Java Users Group) leader and an ESJUG (Espírito Santo Java Users Group) coordinator, both of which are Brazilian JUGs.

Loiane also contributes to the software development community through her blogs, which can be found at [http://loianegroner.com](http://loianegroner.com) (the English version) and [http://loiane.com](http://loiane.com) (the Portuguese-BR version), where she writes about IT careers, Ext JS, Sencha Touch, PhoneGap, Spring Framework, Java, and general development notes and also publishes screencasts.
If you want to keep in touch with her, you can find her on Facebook (https://www.facebook.com/loianegroner) and on Twitter (@loiane).

**Gabriel Manricks** is a full-stack software and web developer, and a writer. He is the CTO at CoinSimple and a staff writer at Nettuts+, where he enjoys learning as well as teaching others. He also freelances in the fields of web consulting, development, and writing.
JavaScript Regular Expressions

Regular expressions are patterns or templates, which allow you to define a set of rules in a natural yet vague way, giving you the ability to match and validate text. They have, more or less, been implemented in nearly every modern programming language.

When working with any type of textual input, you don't always know what the value will be, but you can usually assume (or even demand) the format you are going to receive into your application. These types of situations are exactly when you would create a regular expression to extract and manipulate this input.

In this book, you will learn the basics to get started with a regular expression in JavaScript. We will start with the basics, passing through some special patterns and then, dive into two examples. The first one is validating a web form, and the second one is a very complex pattern to extract information from a log file. For all the examples, we will use a step-by-step approach, which will make it easier to learn and assimilate all the knowledge we've gained from this book.

What This Book Covers

Chapter 1, Getting Started with Regex, presents an introduction about regular expressions in JavaScript. It also shows how to develop the program that will be used to test the regular expressions used in the first three chapters.

Chapter 2, The Basics, covers the main features of regular expressions in JavaScript, which are vague matchers, multipliers, and ranges.

Chapter 3, Special Characters, dives into the special characters patterns of Regex. It covers defining boundaries for a Regex, defining nongreedy quantifiers, and defining Regex with groups.

Chapter 4, Regex in Practice, demonstrates how to develop a web form and validate all its fields using regular expressions functionalities learned since the first chapter.

Chapter 5, Node.js and Regex, explains step by step how to create a simple application using Node.JS to read and parse an Apache log file using Regex. It also demonstrates how to display the information from the log file into a friendly web page to the user.

Appendix, JavaScript Regex Cheat Sheet, presents a summary of the patterns used in regular expressions in JavaScript along with their descriptions, and a list of useful methods to test and create regular expressions.
Regular expressions are special kinds of tools used to represent patterns syntactically. When working with any type of textual input, you don't always know what the value will be, but you can usually assume (or even demand) the format you are going to receive into your application. These types of situations arise when you create a regular expression to extract and manipulate this input.

Consequently, to match a specific pattern requires a very mechanical syntax, since a change in even a single character or two can vastly change the behavior of a regular expression and, as a result, the final outcome as well.

Regular expressions by themselves (or Regex, for short) are not specific to any single programming language and you can definitely use them in nearly all the modern languages straight out of the box. However, different languages have implemented Regex with different feature sets and options; in this book, we will be taking a look at Regex through JavaScript, and its specific implementation and functions.

It's all about patterns

Regular expressions are strings that describe a pattern using a specialized syntax of characters, and throughout this book, we will be learning about these different characters and codes that are used to match and manipulate different pieces of data in a vague sort of manner. Now, before we can attempt to create a regular expression, we need to be able to spot and describe these patterns (in English). Let's take a look at a few different and common examples and later on in the book, when we have a stronger grasp on the syntax, we will see how to represent these patterns in code.
Analyzing a phone number

Let's begin with something simple, and take a look at a single phone number:

123-123-1234

We can describe this pattern as being three digits, a dash, then another three numbers, followed by a second dash, and finally four more numbers. It is pretty simple to do; we look at a string and describe how it is made up, and the preceding description will work perfectly if all your numbers follow the given pattern. Now, let's say, we add the following three phone numbers to this set:

123-123-1234
(123)-123-1234
1231231234

These are all valid phone numbers, and in your application, you probably want to be able to match all of them, giving the user the flexibility to write in whichever manner they feel most comfortable. So, let's have another go at our pattern. Now, I would say we have three numbers, optionally inside brackets, then an optional dash, another three numbers, followed by another optional dash, and finally four more digits. In this example, the only parts that are mandatory are the ten digits: the placing of dashes and brackets would completely be up to the user.

Notice also that we haven't put any constraints on the actual digits, and as a matter of fact, we don't even know what they will be, but we do know that they have to be numbers (as opposed to letters, for instance), so we've only placed this constraint:
Analyzing a simple log file

Sometimes, we might have a more specific constraint than just a digit or a letter; in other cases, we may want a specific word or at least a word from a specific group. In these cases (and mostly with all patterns), the more specific you can be, the better. Let's take the following example:

```
[info] - App Started
[warning] - Job Queue Full
[info] - Client Connected
[error] - Error Parsing Input
[info] - Application Exited Successfully
```

This is an example of some sort of log, of course, and we can simply say that each line is a single log message. However, this doesn't help us if we want to manipulate or extract the data more specifically. Another option would be to say that we have some kind of word in brackets, which refers to the log level, and then a message after the dash, which will consist of any number of words. Again, this isn't too specific, and our application may only know how to handle the three preceding log levels, so, you may want to ignore everything else or raise an error.

To best describe the preceding pattern, we would say that you have a word, which can either be info, a warning, or an error inside a pair of square brackets, followed by a dash and then some sort of sentence, which makes up the log message. This will allow us to capture the information from the log more accurately and make sure our system is ready to handle the data before we send it:

```
[info warning error] - Log Message
```

Analyzing an XML file

The last example I want to discuss is when your pattern relies on itself; a perfect example of this is with something like XML. In XML you may have the following markup:

```
<title>Demo</title>
<size>45MB</size>
<date>24 Dec, 2013</date>
```
We could just say that the pattern consists of a tag, some text, and a closing tag. This isn't really specific enough for it to be a valid XML, since the closing tag has to match the opening one. So, if we define the pattern again, we would say that it contains some text wrapped by an opening tag on the left-hand side and a matching closing tag on the right-hand side:

![Diagram of match tag]

The last three examples were just used to get us into the Regex train of thought; these are just a few of the common types of patterns and constraints, which you can use in your own applications.

Now that we know what kind of patterns we can create, let's take a moment to discuss what we can do with them; this includes the actual features and functions JavaScript provides to allow us to use these patterns once they're made.

**Regex in JavaScript**

In JavaScript, regular expressions are implemented as their own type of object (such as the `RegExp` object). These objects store patterns and options and can then be used to test and manipulate strings.

To start playing with regular expressions, the easiest thing to do is to enable a JavaScript console and play around with the values. The easiest way to get a console is to open up a browser, such as Chrome, and then open the JavaScript console on any page (press the `command + option + J` on a Mac or `Ctrl + Shift + J`).

Let's start by creating a simple regular expression; we haven't yet gotten into the specifics of the different special characters involved, so for now, we will just create a regular expression that matches a word. For example, we will create a regular expression that matches `hello`. 
The RegExp constructor

Regular expressions can be created in two different ways in JavaScript, similar to the ones used in strings. There is a more explicit definition, where you call the constructor function and pass it the pattern of your choice (and optionally any settings as well), and then, there is the literal definition, which is a shorthand for the same process. Here is an example of both (you can type this straight into the JavaScript console):

```javascript
var rgx1 = new RegExp("hello");
var rgx2 = /hello/;
```

Both these variables are essentially the same, it's pretty much a personal preference as to which you would use. The only real difference is that with the constructor method you use a string to create an expression: therefore, you have to make sure to escape any special characters beforehand, so it gets through to the regular expression.

Besides a pattern, both forms of Regex constructors accept a second parameter, which is a string of flags. Flags are like settings or properties, which are applied on the entire expression and can therefore change the behavior of both the pattern and its methods.

Using pattern flags

The first flag I would like to cover is the ignore case or i flag. Standard patterns are case sensitive, but if you have a pattern that can be in either case, this is a good option to set, allowing you to specify only one case and have the modifier adjust this for you, keeping the pattern short and flexible.

The next flag is the multiline or m flag, and this makes JavaScript treat each line in the string as essentially the start of a new string. So, for example, you could say that a string must start with the letter a. Usually, JavaScript would test to see if the entire string starts with the letter a, but with the m flag, it will test this constraint against each line individually, so any of the lines can pass this test by starting with a.

The last flag is the global or g flag. Without this flag, the RegExp object only checks whether there is a match in the string, returning on the first one that's found; however, in some situations, you don't just want to know if the string matches, you may want to know about all the matches specifically. This is where the global flag comes in, and when it's used, it will modify the behavior of the different RegExp methods to allow you to get to all the matches, as opposed to only the first.
Getting Started with Regex

So, continuing from the preceding example, if we wanted to create the same pattern, but this time, with the case set as insensitive and using global flags, we would write something similar to this:

```javascript
var rgx1 = new RegExp("hello", "gi");
var rgx2 = /hello/gi;
```

### Using the rgx.test method

Now that we have created our regular expression objects, let's use its simplest function, the `test` function. The `test` method only returns `true` or `false`, based on whether a string matches a pattern or not. Here is an example of it in action:

```javascript
> var rgx = /hello/;
undefined
> rgx.test("hello");
true
> rgx.test("world");
false
> rgx.test("hello world");
true
```

As you can see, the first string matches and returns `true`, and the second string does not contain `hello`, so it returns `false`, and finally the last string matches the pattern. In the pattern, we did not specify that the string had to only contain `hello`, so it matches the last string and returns `true`.

### Using the rgx.exec method

The next method on the `RegExp` object, is the `exec` function, which, instead of just checking whether the pattern matches the text or not, `exec` also returns some information about the match. For this example, let's create another regular expression, and get `index` for the start of the pattern:

```javascript
> var rgx = /world/;
undefined
> rgx.exec("world !!");
[ 'world' ]
> rgx.exec("hello world");
[ 'world' ]
> rgx.exec("hello");
null
```
As you can see here, the result from the function contains the actual match as the first element (`rgx.exec("world !!")[0];`) and if you `console.dir` the results, you will see it also contains two properties: `index` and `input`, which store the starting index property and complete the `input` text, respectively. If there are no matches, the function will return `null`:

```
> var rgx = /world/;
  undefined
> console.dir(rgx.exec("hello world"));
  Array[1] 0: "world"
  index: 6
  input: "hello world"
  length: 1
  __proto__: Array[0]
< undefined
```

**The string object and regular expressions**

Besides these two methods on the `RegExp` object itself, there are a few methods on the string object that accept the `RegExp` object as a parameter.

**Using the String.replace method**

The most commonly used method is the `replace` method. As an example, let's say we have the `foo foo` string and we want to change it to `qux qux`. Using `replace` with a string would only switch the first occurrence, as shown here:

```
< undefined
> str = "foo foo"
  "foo foo"
> str.replace("foo", "qux")
  "qux foo"
```
Getting Started with Regex

In order to replace all the occurrences, we need to supply a `RegExp` object that has the `g` flag, as shown here:

Using the `String.search` method

Next, if you just want to find the (zero-based) index of the first match in a string, you can use the `search` method:

```
> str = "hello world";
"hello world"
> str.search(/world/);
6
```

Using the `String.match` method

The last method I want to talk about right now is the `match` function. This function returns the same output as the `exec` function we saw earlier when there was no `g` flag (it includes the `index` and `input` properties), but returned a regular `Array` of all the matches when the `g` flag was set. Here is an example of this:

```
< undefined
> var str = "abcabc";
undefined
> console.dir(str.match(/b/));
  ▼ Array[1]
    0: "b"
      index: 1
      input: "abcabc"
      length: 1
        ▶ __proto__: Array[0]
  ▼ undefined
> console.dir(str.match(/b/g));
  ▼ Array[2]
    0: "b"
      1: "b"
      length: 2
        ▶ __proto__: Array[0]
  ▼ undefined
> 
```
We have taken a quick pass through the most common uses of regular expressions in JavaScript (code-wise), so we are now ready to build our `RegExp` testing page, which will help us explore the actual syntax of Regex without combining it with JavaScript code.

## Building our environment

In order to test our Regex patterns, we will build an **HTML** form, which will process the supplied pattern and match it against a string.

I am going to keep all the code in a single file, so let's start with the head of the **HTML** document:

```html
<!DOCTYPE html>
<html lang="en">
<head>
<title>Regex Tester</title>
<link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.css">
<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.0.3/jquery.min.js"></script>
<style>
body{
  margin-top: 30px;
}
.label {
  margin: 0px 3px;
}
</style>
</head>
```

### Downloading the example code

You can download the example code files from your account at [http://www.packtpub.com](http://www.packtpub.com) for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you.

It is a fairly standard document head, and contains a title and some styles. Besides this, I am including the bootstrap **CSS** framework for design, and the jQuery library to help with the **DOM** manipulation.
Next, let's create the form and result area in the body:

```html
<body>
    <div class="container">
        <div class="row">
            <div class="col-sm-12">
                <div class="alert alert-danger hide" id="alert-box"></div>
                <div class="form-group">
                    <label for="input-text">Text</label>
                    <input type="text" class="form-control" id="input-text" placeholder="Text">
                </div>
                <label for="inputRegex">Regex</label>
                <div class="input-group">
                    <input type="text" class="form-control" id="input-regex" placeholder="Regex">
                    <span class="input-group-btn">
                        <button class="btn btn-default" id="test-button" type="button">Test!</button>
                    </span>
                </div>
            </div>
        </div>
    </div>
    <div class="row">
        <h3>Results</h3>
        <div class="col-sm-12">
            <div class="well well-lg" id="results-box"></div>
        </div>
    </div>
</div>
<script>
</script>
```
Most of this code is boilerplate HTML required by the Bootstrap library for styling; however, the gist of it is that we have two inputs: one for some text and the other for the pattern to match against it. We have a button to submit the form (the Test! button) and an extra div to display the results.

Opening this page in your browser should show you something similar to this:

Handling a submitted form

The last thing we need to do is handle the form being submitted and run a regular expression. I broke the code into helper functions to help with the code flow when we go through it now. To begin with, let's write the full-click handler for the submit (Test!) button (this should go where I've inserted the comment in the script tags):

```javascript
//get current values
var text = textbox.val();
var regex = regexbox.val();
```
Getting Started with Regex

//handle empty values
if (text == "") {
    err("Please enter some text to test.");
} else if (regex == "") {
    err("Please enter a regular expression.");
} else {
    regex = createRegex(regex);

    if (!regex) {
        return;
    }

    //get matches
    var results = getMatches(regex, text);

    if (results.length > 0 && results[0] !== null) {
        var html = getMatchesCountString(results);
        html += getResultsString(results, text);
        resultsbox.html(html);
    } else {
        resultsbox.text("There were no matches.");
    }
}

The first four lines select the corresponding DOM element from the page using jQuery, and store them for use throughout the application. This is a best practice when the DOM is static, instead of selecting the element each time you use it.

The rest of the code is the click handler for the submit (Test!) button. In the function that handles the Test! button, we start by clearing the results and errors from the previous run. Next, we pull in the values from the two text boxes and handle the cases where they are empty using a function called err, which we will take a look at in a moment. If the two values are fine, we attempt to create a new RegExp object and we get their results using two other functions I wrote called createRegex and getMatches, respectively. Finally, the last conditional block checks whether there were results and displays either a No Matches Found message or an element on the page that will show individual matches using getMatchesCountString to display how many matches were found and getResultsString to display the actual matches in string.
Resetting matches and errors
Now, let's take a look at some of these helper functions, starting with `err` and `clearResultsAndErrors`:

```javascript
function clearResultsAndErrors() {
  resultsbox.text('"");
  alertbox.addClass("hide").text('"");
}

function err(str) {
  alertbox.removeClass("hide").text(str);
}
```

The first function clears the text from the results element and then hides the previous errors, and the second function un-hides the alert element and adds the error passed in as a parameter.

Creating a regular expression
The next function I want to take a look at is in charge of creating the actual RegExp object from the value given in the textbox:

```javascript
function createRegex(regex) {
  try {
    if (regex.charAt(0) == "/") {
      regex = regex.split("/");
      regex.shift();
      var flags = regex.pop();
      regex = regex.join("/");

      regex = new RegExp(regex, flags);
    } else {
      regex = new RegExp(regex, "g");
    }
    return regex;
  } catch (e) {
    err("The Regular Expression is invalid.");
    return false;
  }
}
```
If you try and create a `RegExp` object with flags that don't exist or invalid parameters, it will throw an exception. Therefore, we need to wrap the `RegExp` creation in a `try/catch` block, so that we can catch the error and display an error for it.

Inside the `try` section, we will handle two different kinds of `RegExp` input, the first is when you use forward slashes in your expressions. In this situation, we split this expression by forward slashes, remove the first element, which will be an empty string (the text before it is the first forward slash), and then pop off the last element which is supposed to be in the form of flags.

We then recombine the remaining parts back into a string and pass it in along with the flags into the `RegExp` constructor. The other case we are dealing with is where you wrote a string, and then we are simply going to pass this pattern to the constructor with only the `g` flag, so as to get multiple results.

### Executing RegExp and extracting its matches

The next function we have is for actually cycling through the `regex` object and getting results from different matches:

```javascript
function getMatches(regex, text) {
    var results = [];
    var result;

    if (regex.global) {
        while((result = regex.exec(text)) !== null) {
            results.push(result);
        }
    } else {
        results.push(regex.exec(text));
    }

    return results;
}
```

We have already seen the `exec` command earlier and how it returns a `results` object for each match, but the `exec` method actually works differently, depending on whether the global flag (`g`) is set or not. If it is not set, it will constantly just return the first match, no matter how many times you call it, but if it is set, the function will cycle through the results until the last match returns `null`. In the function, the global flag is set, I use a while loop to cycle through `results` and push each one into the `results` array, whereas if it is not set, I simply call `function` once and push only if the first match on.
Next, we have a function that will create a string that displays how many matches we have (either one or more):

```javascript
function getMatchesCountString(results) {
    if (results.length === 1) {
        return "<p>There was one match.</p>";
    } else {
        return "<p>There are " + results.length + " matches.</p>";
    }
}
```

Finally, we have a function, which will cycle through the results array and create an HTML string to display on the page:

```javascript
function getResultsString(results, text) {
    for (var i = results.length - 1; i >= 0; i--) {
        var result = results[i];
        var match = result.toString();
        var prefix = text.substr(0, result.index);
        var suffix = text.substr(result.index + match.length);
        text = prefix
            + '<span class="label label-info">'
            + match
            + '</span>'
            + suffix;
    }
    return '<h4>' + text + '</h4>";
}
```

Inside the function, we cycle through a list of matches and for each one, we cut the string and wrap the actual match inside a label for styling purposes. We need to cycle through the list in reverse order as we are changing the actual text by adding labels and also so as to change the indexes. In order to keep in sync with the indexes from the results array, we modify the text from the end, keeping the text that occurs before it, the same.
Testing our application

If everything goes as planned, we should now be able to test the application. For example, let’s say we enter the Hello World string as the text and add the l pattern (which if you remember will be similar to entering /l/g into our application), you should get something similar to this:

```
Text
Hello World

Regex
l

Results
There are 3 matches.
Hello World
```

Whereas, if we specify the same pattern, though without the global flag, we would only get the first match:

```
Text
Hello World

Regex
/l/

Results
There was one match.
Hello World
```
Of course, if you leave out a field or specify an invalid pattern, our error handling will kick in and provide an appropriate message:

![Error Message]

With this all working as expected, we are now ready to start learning Regex by itself, without having to worry about the JavaScript code alongside it.

**Summary**

In this chapter, we took a look at what a pattern actually is, and at the kind of data we are able to represent. Regular expressions are simply strings that express these patterns, and combined with functions provided by JavaScript, we are able to match and manipulate user data.

We also covered building a quick RegExp builder that allowed us to get a first-hand look at how to use regular expressions in a real-world setting. In the next chapter, we will continue to use this testing tool to start exploring the RegExp syntax.
Where to buy this book

You can buy JavaScript Regular Expressions from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. 

Click here for ordering and shipping details.