

The Regular Expressions Book – RegEx for JavaScript Developers Full Book ??

Source : [The Regular Expressions Book - RegEx for JavaScript Developers Full Book](#)

The Regular Expressions Book – RegEx for JavaScript Developers [Full Book]

If you want to master regular expressions and understand how they work in JavaScript, this book's for you.

Regular expressions can be intimidating when you first encounter them. When I started learning to code, I gave up on regular expressions twice.

While that was partly because I was intimidated by regular expressions at first, the tutorials and courses I used never taught them in a way I could understand.

In fact, before some tutorials start teaching regex, they complain about regex and how tough they can be. And there's no better way to discourage a learner than that.

In this book, you won't just see how to use regex in a regex testing tool like **regexpal** or **regex101**. You'll also see how they work in JavaScript. This is what many courses and tutorials tailored for regex in JavaScript lack. As you see how they work using a regex tester, you'll also see how they work in JavaScript.

You can also apply what you learn in this book to other programming languages like Python, PHP, and so on. All you need to do is to know about how the regex engine of that language works. You'll also need to understand the methods and functions the language uses for working with regular expressions.

To get the most out of this book, make sure you read it in order because each chapter builds upon the previous ones. I have also arranged the chapters according to how difficult they are. So, you will find simpler concepts first and more advanced concepts later.

Happy reading!

Chapter 1: Introduction to Regular Expressions

What are Regular Expressions?

You might see this written as regular expressions, regex, or RegExp – but all refer to the same thing.

Regex are a sequence of characters for matching a part of a string or the whole string. Matching strings with regular expressions might require more than just "characters". Many times, you will need to use a special set of characters called "metacharacters" and "quantifiers".

Because regular expressions are a powerful tool, you can use them to do much more than just "matching strings" when you combine regex with programming languages.

Almost all the main programming languages of the modern era have built-in support for regular expressions. Some programming languages might even have specific libraries that help you work more conveniently with regex.

Apart from using regular expressions in programming languages, other tools that let you use regular expressions are:

- **Text Editors and IDEs:** for search and search and replace in VS Code, Visual Studio, Notepad++, Sublime Text, and others.
- **Browser Developer Tools:** mostly in-browser search (with extensions or add-ons) and search within the developer tools.
- **Database Tools:** for data mining.
- **RegEx Testers:** you can paste in text and write the regular expressions to match them – which is a very good way to learn regular expressions. This book explores that option quite a bit.

A Brief History of Regular Expressions

Regular expressions have a rich and fascinating history that has already spanned more than seven decades. This history continues to evolve alongside the development of computer science and programming languages.

The concept of regular expressions traces back to the 1950s. American mathematician Stephen Cole Kleene introduced them as a notation for defining patterns in formal languages. Kleene's work also formed the foundation for theoretical computer science.

In the early 1960s, the first implementations of regular expressions emerged. Ken Thompson, a computer scientist at Bell Labs, developed a text editor named **QED** that utilized regular

expressions for pattern matching. QED's capabilities provided a way to search and manipulate texts more efficiently.

The concept gained further popularity when Thompson and Dennis Ritchie created the Unix operating system in the early 1970s.

They incorporated regular expressions into various Unix utilities, most notably the **ed text editor** and later the **sed stream editor**. These tools allowed users to perform complex text manipulation tasks, significantly enhancing the efficiency and power of text processing.

In 1973, Thompson collaborated with Alfred Aho and Peter Weinberger to develop a new tool called **grep** (global regular expression print) as part of the Unix toolkit.

Grep allowed users to search files for specific patterns using regular expressions. The simplicity and effectiveness of grep made it a widely adopted tool. It also established regular expressions as a standard feature in Unix-based systems.

As computer systems and programming languages evolved, regular expressions became integrated into various software development environments. In the late 1970s, the AWK programming language was created. AWK inspired Larry Wall to create Perl and make it available to the public in 1987.

Wall recognized the value of regular expressions for text manipulation and integrated regex into Perl.

Perl's integration of regular expressions into its syntax made it a popular language for text matching and data extraction tasks. This integration formed the foundation of **PCRE** (Perl-compatible regular expressions), a flavor and library of regular expressions you can use in some programming languages such as Perl, Python, PHP, Java, and others.

Regular expressions continued to evolve and find applications beyond Unix and Perl. In the 1980s, the International Organization for Standardization (ISO) developed the POSIX standard, which included a specification for regular expressions. This standardization ensured compatibility and consistency across different implementations and systems.

With the rise of the internet and the World Wide Web in the 1990s, regular expressions found widespread use in web development and data processing. They became an essential component of many scripting languages, providing developers with powerful tools for text processing, form validation, and data extraction from web pages.

For example, JavaScript had always had a version of PCRE built in for working with regular expressions. But by 1999, with the release of ECMAScript, the `RegExp()` constructor was introduced. This gave JavaScript developers the ability to start using regular expressions directly in their code, in the JavaScript way.

In the early 2000s, tools and libraries specifically focused on regular expressions emerged, making it easier for developers to work with them. Libraries like PCRE (Perl Compatible Regular

Expressions) provided enhanced features and better performance, further expanding the usage and capabilities of regular expressions.

Today, regular expressions are an integral part of programming languages and text-processing tools like your code editor. They are supported by almost all major programming languages, including Java, C#, Ruby, and PHP.

Integrated development environments (IDEs) and code editors like Visual Studio, VS Code, and Notepad++ also now include regex-based search and search and replace functionalities, simplifying the process of finding and manipulating texts in code.

The history of regular expressions demonstrates their evolution from theoretical concepts to practical tools that have revolutionized text processing and pattern matching.

From the early developments at Bell Labs and Unix to their integration into popular programming languages, regular expressions have become an essential tool in the hands of developers and system administrators. Regex empowers them to handle complex text-based tasks efficiently.

With the ongoing advancements in computing and the continuous demand for efficient text processing, regular expressions will likely remain a fundamental part of the technology landscape for years to come.

What are the Uses of Regular Expressions?

Regular expressions are quite versatile and flexible. This makes it possible to apply them to various tasks in various domains such as computer programming, data processing, text editing, and web development.

Those applications and uses include but are not limited to the following:

String Matching: This is one of the most common ways developers use regular expressions. This is also a good way to learn regular expressions.

You can paste some texts into a regex engine and write the regex to match a part of the text or the whole text. You can also search for strings that contain specific character sequences, start or end with certain characters, or match complex patterns.

This makes regular expressions valuable for tasks like searching for keywords, validating input against specific patterns, or filtering data based on string patterns

Password Strength Validation: You can use regular expressions for validating the strength of passwords in websites and applications.

By defining a set of rules using regular expressions, developers can enforce specific password requirements, such as a minimum number of characters, a combination of uppercase and lowercase letters, numbers, and special characters.

Form Validation: Validating inputs of a form or standalone inputs is another popular way developers use regular expressions.

Regular expressions provide a concise and efficient way to ensure that input data follows specific patterns or formats. Whether it's validating usernames, email addresses, phone numbers, credit card numbers, postal codes, or other inputs, regular expressions can help you enforce validation rules and maintain data integrity.

Text Search and Manipulation: Regular expressions excel at searching for specific patterns within text and performing manipulations based on those matches. They are a powerful tool for tasks such as data mining, log analysis, and text processing.

Whether you need to find occurrences of particular words or phrases, extract structured data from text, analyze content, or perform string matching, regular expressions offer efficient pattern-matching capabilities.

Working with URLs and URIs: Since URLs and URIs are an integral part of web development, regular expressions can help in validating, parsing, and manipulating them. This enables developers to ensure the correctness and structure of web addresses, validate whether a string is a valid URL, and help extract specific components such as the domain, path, query parameters, or fragments.

This functionality is particularly useful in tasks like URL routing, rewriting, or extracting data from query parameters.

Search and Replace in IDEs and Text Editors: Regular expressions offer sophisticated search capabilities. This enables developers to locate specific patterns (such as words with specific prefixes or sequences of characters) and then replace the matches with a specified text. This is built into modern text editors like VS Code and Notepad++.

Data Extraction and Scraping: Regular expressions play a significant role in data extraction and web scraping. They allow developers to extract specific information from unstructured or semi-structured text by defining patterns to match desired data.

They are also valuable when extracting data from sources like HTML or XML documents, as they enable efficient retrieval of information based on defined patterns.

Syntax Highlighting: Regular expressions are commonly used in IDEs and text editors to provide syntax highlighting. This ends up helping users to visually distinguish different parts of a code or document by assigning colors or formatting to keywords, strings, comments, and other language-specific constructs.

Regular expressions are used to identify and match these language-specific patterns, making code more readable and enhancing the overall editing experience.

Flavors of Regular Expressions

The term "flavors of regular expressions" refers to the specific implementation and syntax variations of regular expressions in different programming languages, libraries, or tools.

While the core concept of regular expressions remains the same, the details of how regular expressions are written and interpreted can vary between different environments.

Each flavor of regular expressions may have its own set of metacharacters, syntax rules, and additional features beyond the basic functionality.

These differences can include variations in the syntax for character classes, metacharacters, capturing groups, and assertions, as well as additional capabilities like named capturing groups, look-ahead, and look-behinds.

There are many flavors of regular expressions available today. Some of them are:

- **Basic Regular Expressions (BRE):** this flavor is commonly found in Unix tools such as **sed** and **grep**. It uses a limited set of metacharacters and features. The wildcard (`.`) and zero or more (`*`) metacharacters are available in it.
- **Extended Regular Expressions (ERE):** ERE is an extension of BRE. It provides additional metacharacters and features. In addition to the metacharacters available in BRE, ERE introduces features like grouping with parentheses (`()`), alternation with the pipe symbol (`|`), and the use of curly braces (`{ }`) to specify repetition ranges.
- **Perl-Compatible Regular Expressions (PCRE):** PCRE is a popular flavor supported by various programming languages such as Perl, Python, PHP, and JavaScript. PCRE extends the basic regular expression syntax with powerful features like lookahead and look-behind assertions, backreferences, non-capturing groups, and the use of `\b` for word boundaries.
- **JavaScript Regular Expressions:** JavaScript has its regular expression flavor which is similar to PCRE but with a few differences. It supports basic features like character classes with square brackets (`[]`), metacharacters (`*`, `+`, `?`, and others), and capturing groups (`()`). JavaScript also provides additional features like the global flag `/g` to perform multiple matches, and the ignore case flag `/i` for case-insensitive matching.
- **Python Regular Expressions:** Python's `re` module implements a flavor that is similar to PCRE but with a few variations. It supports features such as character classes `[]`, metacharacters (`*`, `+`, and `?`), and capturing groups (`()`). The `re` module also has a unique raw string syntax (`r' '`) to simplify working with backslashes.

It's important to be aware of the flavor of regular expressions you are using when working with regular expressions in different programming languages or tools. This ensures that you use the correct syntax and take advantage of any unique features or capabilities provided by that particular flavor.

N.B.: Don't bother so much about the metacharacters (and quantifiers) mentioned in this part. You will see them in action in chapter 5 of this book.

Tools for Working with Regular Expressions

Regular expression tools are the programming languages, libraries and frameworks, command line utilities, online regex testers, text editors and IDEs, and applications designed to help you create, test, and apply regular expressions in your day-to-day work life.

There are many tools available for working with regular expressions. Let me take you through them under regex testers, programming languages, libraries, text editors and IDEs, and command line tools.

RegEx Testers

RegEx testers are the online testing environments specifically built for creating and testing regular expressions against some test strings. Examples include regex101.com, regexr.com, and regexpal.com.

The UIs of these regex testers usually have an input for the regular expressions you want to write, and another for the text you want to test the regex against.

This is how the UI of regexpal.com looks:

regexpal-ui

More advanced ones like regex101.com let you select the flavor of regular expressions you want to work with, an explanation of the regex, and match information.

Here's what the UI of regex101.com looks like:

regex101-ui-1

One of the good things about these online regex testers is that they are helpful for learning regular expressions. A lot of them provide real-time matching and cheatsheets you can quickly look at. Many devs who use regex have used them.

Apart from learning, you can also use them by creating your regex with them and pasting them into wherever you want to use the regex. This is how I create my regex.

Programming Languages

Almost all modern programming languages have built-in support for regular expressions. And so they all have methods for creating and testing regular expressions.

For example, JavaScript has the `RegExp()` constructor for working with regular expressions, Python has the `re` module, Java has the `java.util.regex` package, and Perl has regex built into it directly.

Libraries and Frameworks

Many programming languages have standalone libraries and frameworks that make it easier to create regular expressions.

There is `XRegExp` for JavaScript, PCRE (Perl Compatible Regular Expressions) for Perl, Go-Reuse for Golang, and Verbal Expressions, a cross-platform regex library.

Text Editors and IDEs

Many text editors and IDEs such as VS Code, Visual Studio, Notepad++, Atom, Sublime Text, IntelliJ IDEA, and others have built-in support for regular expressions.

The commonest thing developers use this for is search, and search and replace. Also, the syntax highlighting in those text editors and IDEs is often implemented with regular expressions.

Command Line Tools

Unix command line tools like `grep` and `sed` allow you to perform regex operations on text files and streams. With this, you can search, filter, and manipulate multiple files.

Using these Unix tools, options for customizing search behaviors and customizing complex text transformations are also available to you.

Basic Concepts of Regular Expressions

The basic concepts and syntax of regular expressions are the building blocks involved in creating, testing, and applying patterns for searching, matching, and manipulating strings.

This includes concepts like **literal characters**, **metacharacters**, **quantifiers**, **character classes**, **anchors and boundaries**, and **escape characters**. The more advanced ones are **groupings**, **backreferences**, **look-ahead assertions**, and **look-behind assertions**.

Regular expressions users utilize many of these concepts to construct efficient regular expressions for working with text. On many occasions, the basic ones are enough. But if you want to create more advanced regular expressions, then the more advanced ones will also be useful for you.

This book won't leave any of the concepts behind. I will show you how you can utilize them in regex testers and how you can use them in JavaScript since that's what this book is meant for.

Chapter 2: How to Match Literal Characters and Character Sets in Regular Expressions

What are Literal Characters in Regular Expressions?

Literal characters are characters you can match as they appear in a test string. They could be letters, numbers, spaces, or even symbols. In other words, they are non-special characters that represent themselves.

This means if you want to match literal characters, you should construct your regex pattern in the same way as the test string appears.

For example, if you want to match the word `hello`, your regex pattern can be `hello`. And if you want to match the `h` in the word `hatch`, all you need as the pattern is `h`.

This `h` would match the first occurrence of the letter `h` in the test string `hatch`. If you want it to match the other letter `h` as well, you need the "g" flag, or global flag. You will learn about the flags and modifiers in the next chapter of this book.

That is not the case for some symbols, though. That's because some symbols are special characters of regular expressions (metacharacters and quantifiers). So, if you want to match those characters, you have to escape them with a backslash (`\`). This book will also teach you all you need to know about metacharacters because there's a whole chapter for them.

How to Match Literal Characters in RegEx Testers

Provided you want to match the word `hello`, then `hello` should be your regex pattern:

hello-match-1

If you want to match the text `freeCodeCamp`, you can construct your regex to be `freeCodeCamp`:

fcc-match-1

So, what if you want to match `hello freeCodeCamp`? Then you just use `hello freeCodeCamp` as the pattern:

hello-fcc-match

If you want to match the letter `e` in the text `freeCodeCamp`, `e` is the pattern to use:

e-in-fcc

And if you want to match `h` in the text `hatch`, `h` is the pattern you should use:

h-in-hatch

You can see that in the text `freeCodeCamp`, the other `e`s after the first occurrence were not returned as matches – same with the last `h` in the word `hatch`. You will learn how to match every occurrence of a letter in a text in the next chapter.

Character Set Matching

A character set, also called character class, is a set of characters that will successfully match a certain character in a test string. This set of characters is enclosed in square brackets.

For instance, the pattern `[abc]` will match any of `a`, `b`, and `c`, while `[xyz]` will match any of `x`, `y`, and `z`.

Here are some examples of character sets and what they do:

- `[abc]`: matches either `a`, `b`, or `c`
- `[aeiou]`: matches any vowel character
- `[a-z]`: matches any lowercase letter from `a` to `z`
- `[A-Z]`: matches any uppercase letter from `A` to `Z`
- `[0-9]`: matches any digit from 0 to 9

Inside the square brackets, you don't need to escape metacharacters because they lose their special meaning. The only symbol that has a meaning in the square brackets is a hyphen (`-`), which you can use to specify ranges, as I have done with some examples of character sets.

You will also learn about ranges in this book. On some occasions, a backslash `\` does not lose its special meaning in a character set.

As with literal character matching, only the first occurrence of the character set will return as a match, every other occurrence will be ignored. In the next chapter, you will learn how to match multiple occurrences of a character with the `g` flag.

Here's how each of the above character sets works in a regex testing tool:

`[abc]`:

-abc--match

`[aeiou]`:

vowels-match

`[a-z]`:

lcase-set-match

`[A-Z]`:

ucase-set-match

`[0-9]`:

number-set-match

You can also define your unique character class based on what you want. Character sets are useful when you want to match some characters in a particular position in a text.

For instance, the pattern `br[ao]ke` will match both `brake` and `broke`:

brakebroke-match

The pattern `gr[ae]y` will match both `gray` and `grey`:

graygrey-match

N.B.: I turned on the `g` flag so you can see all the matches, and how powerful character sets are. We will take a look at the `g` and other flags in the next chapter.

Since there are always multiple ways of doing the same thing in programming, there are also certain character sets called "shorthand character sets" that you can use instead of character sets.

Since these shorthand character sets are a subset of metacharacters, you will learn about them under the chapter dedicated to metacharacters.

Chapter 3: Regular Expressions Flags

Also called modifiers, flags are special characters you can place at the end or within a regular expressions pattern to alter its default behavior.

JavaScript developers tend to refer to these characters as "flags", but in Python they are used interchangeably.

In Python, you can place flags within a regex pattern, but in JavaScript, flags are always placed at the end of the regex pattern.

Here are the flags you can use in regular expressions:

- `global` flag
- `case insensitive` flag
- `multi-line` flag
- `single-line` flag
- `unicode` flag
- `sticky` flag

In many regex engines, you can turn on any flag you want to use. In regex101.com, you can turn on a flag by clicking on the slash symbol (/) right inside the pattern input:

turn-on-a-flag-101

You can then select any flag you want to use:

select-flag-101

N.B.: If the flavor of regex you selected in regex101.com is not ECMAScript, the set of flags presented to you might be different.

If you are using regexpal.com, click on "flags" above the regex pattern input:

turn-on-flag-pal

Select any flag you want by clicking on it:

select-flag-pal

Now, let's take a detailed look at each of the regex flags and how they work in a regex engine.

The global Flag

The global flag is denoted by the letter g. With it, you get to perform a global match with your pattern.

Remember in the previous chapter of this book, some patterns I defined stopped when they found the first match, even if there were more. That's because by default, regular expressions only find the first match in a text. But with the g flag, all occurrences of the match are returned.

Another good thing about using the g flag is that you can iterate over the matches you get with the pattern in JavaScript. The iteration continues until there's nothing to match. You will learn about multiple ways you can iterate over matches soon.

To let you see how the g flag works, I'll use the hatch and freeCodeCamp examples from the previous chapter.

If you want to match the letters h in the word hatch with the pattern h, both the first and the last h's will be returned as matches as long as you have the g flag on:

h-in-hatch-g-match

And if you want to match e in freeCodeCamp with the pattern e and you turn on the g flag, the second and third e's are returned as a match too:

e-in-fcc-g-match

The `case-insensitive` Flag

The `case insensitive` flag is denoted by `i`. As the name implies, it lets you perform case-insensitive matching.

By default, regular expressions perform case-sensitive matching. But with the `i` flag you can perform case-insensitive matching, so you won't bother about casing in your patterns.

With this, uppercase or lowercase will be ignored. That means `Hello` and `hello` will be treated as the same thing:

hello-insensitive

`freeCodeCamp` and `freecodecamp` are treated the same, too:

fcc-insensitive

`RegExp` and `regex` are also the same thing:

regex-insensitive

Another thing is that if you're using a character class, for example `[a-z]`, it would match uppercase letters too if you turn on the `case-insensitive` flag.

So, the pattern `[a-z]` also matches uppercase letters with the `case-insensitive` flag turned on:

carset-insensitive

The `multi-line` and `single-line` Flags

Denoted by `m`, the `multi-line` flag tells the regular expressions engine that the test string is more than one line. Since the `multi-line` flag influences the behavior of the start and end anchor metacharacters (`^` and `$`), you'll learn more about it under the anchors and word boundaries chapter.

The `single-line` flag is denoted by `s`. Just like the `multi-line` flag, the `single-line` flag also works with a metacharacter called the wildcard (`.`). You will see the `single-line` flag in action under the chapter for metacharacters.

The `Unicode` Flag

The Unicode flag enables full Unicode matching in the regular expressions engine that supports it. It is denoted by `u`.

By default, JavaScript and many other programming languages treat strings as a sequence of 16-bit code units. With the `u` flag, regex patterns can match against Unicode code points instead of code units. This allows handling characters like emojis, certain symbols, and characters from non-Latin scripts. So, when you set the flag, it modifies the behavior of certain escape sequences and metacharacters to work with regular expressions.

For example, the escape sequence `\u{1F602}` will match the literal character `u{1F602}` if you don't turn on the `u` flag:

u-flag-literal-match

But if you turn on the `u` flag, the same pattern matches the face with tears emoji:

u-flag-emoji-match

That is one way to match emojis and other Unicode characters. Take the Unicode of the emoji and put the hexadecimal in curly braces, then precede the two with `\u`.

For instance, the Unicode of growing heart is `U+1F497`, the pattern to match it would be `\u{1F497}`:

growing-heart-match

You will see more examples of how the flag works in the chapter on how to use regular expressions in JavaScript.

The `sticky` Flag

The sticky flag is denoted by `y`. It's a feature of JavaScript regular expressions implemented in ECMAScript 6. The `y` flag limits matching to the current position in the string, which you can specify with the `lastIndex` property of the `RegExp()` constructor.

When you use the `y` flag, it uses the `lastIndex` property to determine where the next search will start. The pattern matches only if it occurs exactly at the `lastIndex` position or at the beginning of the string.

Unlike the global (`g`) flag, the `y` flag does not find all matches but stops after the first successful match.

In a regex engine like regex101.com, the `y` flag usually anchors to the start of the test string and stops there:

anchor-flag-match

Since the `y` flag typically works with the `lastIndex` property of JavaScript regular expressions, we will look at more examples in the chapter on how to use regular expressions in JavaScript – specifically when we look at the `sticky` of the regular expressions constructor.

You can also combine multiple flags to write more complex syntax. For example, you can use the `g` flag with the `i` flag for global and case-insensitive matching:

g-and-i-flag

Chapter 4: How to Use Regular Expressions in JavaScript

How to Create Regular Expressions in JavaScript

There are two ways you can create regular expressions in JavaScript. The first is with **regex literal syntax** and the second is with the `RegExp()` constructor.

To create a regular expression with the regex literal syntax, you have to enclose the pattern inside two forward slashes (`/`) like this:

```
/regex pattern/
```

If you want to use one or more flags, it has to be after the second slash:

```
/regex pattern/flag
```

Depending on your use case, you might have to assign the regex to a variable:

```
const regex = /regex pattern/flag
```

The flag could be any of the flags available in the JavaScript regular expressions engine.

If you want to create regular expressions with the `RegExp()` constructor, you have to use the `new` keyword, then put the pattern and the flag inside the `RegExp()` brackets.

This is what the syntax looks like:

```
const regex = new RegExp("regex pattern", "flag");
```

Since `RegExp()` is a constructor, there are some methods and properties available in it with which you can work with regular expressions. Whether you create your pattern with the literal syntax `//` or the `RegExp()` constructor, the methods and properties are available for it.

Methods of the `RegExp()` Constructor

The methods of the `RegExp()` constructor are defined on the `RegExp.prototype`. You can quickly check the methods (and properties) by typing `RegExp().__proto__` and hitting `ENTER` in your browser console. These methods include `test()`, `exec()`, and `toString()`.

Apart from those three, some methods take regular expressions as a parameter. But it is better to discuss them under "string methods for working with regular expressions" because, at their core, they are string methods that take regular expressions as a parameter.

Let's take a look at what `test()`, `exec()`, and `toString()` do.

The `test()` Method

The `test()` method tests for a match between a regular expression and the test string and returns a boolean as the result. If there's a match, it returns `true`, and if there's no match, it returns `false`.

In the example below, there's a match for the pattern `/freeCodeCamp/`:

```
const re = /freeCodeCamp/;
const testStr =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp.";

console.log(re.test(testStr)); //true
```

But in the example below, there's no match for the pattern `/fcc/`, so the `test()` method returns `false`:

```
const re = /fcc/;
const testStr =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp.";

console.log(re.test(testStr)); //false
```

Apart from testing random patterns against a string, the `test()` method can be useful in form validation.

The `exec()` Method

The `exec()` method executes a search for a match in a test string and returns an array containing a piece of detailed information about the first match. If there's no match, it returns `null`.

That detailed information contains the **first match, the index of the match, captured groups** (if any), and the **length**.

Here's an example:

```
const re = /freeCodeCamp/;
const testStr =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp.";

console.log(re.exec(testStr));
```

And here's a screenshot of the result:

exec-res

If you want to make the `exec()` method return all the matches, you can use the `g` flag on the pattern and then loop through with a `while` loop:

```
const re = /freeCodeCamp/g;
const testStr =
  "freeCodeCamp is a great place to start learning to code from scratch. freeCodeCamp doesn't
  charge you any money, that's why it's called freeCodeCamp.";

let match;

while ((match = re.exec(testStr)) !== null) {
  console.log(match[0]);
}
```

Here's what the result looks like in the console:

exec-multi-res

You can go further by accessing the index of the matches this way:

```
const re = /freeCodeCamp/g;
const testStr =
  "freeCodeCamp is a great place to start learning to code from scratch. freeCodeCamp doesn't
  charge you any money, that's why it's called freeCodeCamp.";

let match;

while ((match = re.exec(testStr)) !== null) {
```

```
console.log(match[0]);

// Access the indices of the matches
console.log(match.index);
}
```

exec-indices-res

If there's no match, `exec()` returns null:

```
const re = /fcc/;
const testStr =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp.";

console.log(re.exec(testStr)); //null
```

The `toString()` Method

The `toString()` method converts a regex pattern to a string. In JavaScript, the `toString()` method is in every object. Regular expressions are treated as an object behind the scenes, that's why you can create them with the `new` keyword.

Using this method on a regex pattern converts the pattern to a string:

```
const pattern = /freeCodeCamp/;
const strPattern = pattern.toString();

console.log(strPattern, typeof strPattern); // /freeCodeCamp/ string
```

Even if you create the pattern with the `RegExp()` constructor, you get the result the same way:

```
const pattern = new RegExp('freeCodeCamp');
const strPattern = pattern.toString();

console.log(strPattern, typeof strPattern); // /freeCodeCamp/ string
```

And if you have a flag in the pattern, it would be returned as a part of the string:

```
const pattern = /freeCodeCamp/gi;
const strPattern = pattern.toString();

console.log(strPattern, typeof strPattern); // /freeCodeCamp/gi string
```

Properties of the `RegExp()` Constructor

The properties of the `RegExp()` constructor are defined on the `RegExp.prototype`. They include:

- `RegExp.prototype.global`
- `RegExp.prototype.source`
- `RegExp.prototype.flags`
- `RegExp.prototype.multiline`
- `RegExp.prototype.ignoreCase`
- `RegExp.prototype.dotAll`
- `RegExp.prototype.sticky`
- `RegExp.prototype.unicode`

In short, there are the `global`, `source`, `flags`, `multiline`, `ignoreCase`, `dotAll`, `sticky`, and `unicode`.

Most of the properties check whether a certain flag is used or not. Let's take a look at how each of the properties works.

The `global` Property

The `global` property checks whether the `g` flag is used with a regex pattern or not. If the pattern has the `g` flag, it returns `true`, otherwise it returns `false`.

Remember the `global` (`g`) flag indicates that the regex pattern should not just return the first match but all the matches.

Here's how the `global` property works in code:

```
const re1 = /freeCodeCamp/g;
const re2 = /freeCodeCamp/;
const re3 = new RegExp('freeCodeCamp');
const re4 = new RegExp('freeCodeCamp', 'g');

console.log(re1.global); //true
console.log(re2.global); //false
console.log(re3.global); //false
console.log(re4.global); //true
```

The `flag` Property

The `flag` property returns the flags you use in the regex pattern in alphabetical order. That is, `g` before `i`, `i` before `m`, `m` before `y`, and so on.

In the code below, you can see that the `g` flag comes before `i`, and `m` comes before `y`:

```
const re1 = /freeCodeCamp/gi;
const re2 = new RegExp('freeCodeCamp', 'my');

console.log(re1.flags); //gi
console.log(re2.flags); //my
```

The `source` Property

The `source` property returns the regex pattern as a string. So, it acts like the `toString()` method.

The difference between the `source` property and the `toString()` method is that the `source` property excludes the flag you use with the pattern. Also, the `source` property does not show the literal forward slashes you use for creating the regex.

In the code below, you can see the forward slashes don't get printed, the flags are omitted too, and the `type` is a string:

```
const re1 = /freeCodeCamp/gi;
const re2 = new RegExp('freeCodeCamp', 'my');

const re1Source = re1.source;
const re2Source = re2.source;

console.log(re1Source, typeof re1Source); // freeCodeCamp string
console.log(re2Source, typeof re2Source); // freeCodeCamp string
```

The `multiline` Property

The `multiline` flag is another boolean property of the `RegExp()` constructor. It specifies whether the `multiline` flag is used with the pattern or not by returning `true` or `false`.

Remember the `multiline` (`m`) flag indicates that the test string should be treated as a text that has more than one line.

Here's how the `multiline` property works in action:

```
const re1 = /freeCodeCamp/gi;
const re2 = new RegExp('freeCodeCamp', 'my');

const re1Source = re1.multiline;
const re2Source = re2.multiline;

console.log(re1Source); //false
```

```
console.log(re2Source); // true
```

The ignoreCase Property

The `ignoreCase` property specifies whether the case-insensitive flag (`i`) is used in the regex pattern. It returns `true` if you use the `i` flag and `false` if you don't use it.

```
const re1 = /freeCodeCamp/i;
const re2 = /freeCodeCamp/;
const re3 = new RegExp('freeCodeCamp', 'i');
const re4 = new RegExp('freeCodeCamp');

console.log(re1.ignoreCase); //true
console.log(re2.ignoreCase); // false
console.log(re3.ignoreCase); // true
console.log(re4.ignoreCase); // false
```

The unicode Property

The `unicode` property helps you check whether the Unicode (`u`) flag is used in the regex pattern or not. If it finds the `u` flag, it returns `true`, otherwise it returns `false`.

```
const re1 = /\u{1F1F3}\u{1F1EC}/u; //matches the Nigerian flag emoji
const re2 = /\u{1F1F3}\u{1F1EC}/;
const re3 = new RegExp('\u{1F1F3}\u{1F1EC}', 'u');
const re4 = new RegExp('\u{1F1F3}\u{1F1EC}');

console.log(re1.unicode); //true
console.log(re2.unicode); // false
console.log(re3.unicode); // true
console.log(re4.unicode); // false
```

The sticky Property

The sticky property indicates whether the sticky (`y`) flag is set in the regular expression or not. Even though that's what it does, it's still a bit tricky to understand because of the `lastIndex` property.

When the `y` flag is set, the regex engine in use will attempt to match the pattern starting at the exact position specified by the `lastIndex` property (without using the `g` flag). If a match is found, the `lastIndex` property is updated to the position immediately after the end of the match.

To help you understand that better, here's a code snippet with comments:

```
const re = /xyz/y;
const str = 'xyzxyz';

re.lastIndex = 0;
console.log(re.test(str)); // true – there's a match at index 0 to 2
console.log(re.lastIndex); // 3

re.lastIndex = 1;
console.log(re.test(str)); // false – no match at the specified index
console.log(re.lastIndex); // 0 – resets to 0 because there's no match at the specified index

re.lastIndex = 3;
console.log(re.test(str)); // true – there's a match at index 3 to 5
console.log(re.lastIndex); // 6

re.lastIndex = 6;
console.log(re.test(str)); // false
console.log(re.lastIndex); // 0 – resets to 0 because there's no match at the specified index
```

N.B.: The `dotAll` property works with the wildcard (`.`) metacharacter. Due to that, you will see how it works in detail in the chapter on metacharacters. Also, `hasIndices` works with captures. So, you will see how to use it under the chapter on grouping and capturing.

String Methods for Working with Regular Expressions

JavaScript provides some inbuilt methods for working with strings. Some of these methods take regular expressions as a parameter. These methods include `match()`, `matchAll()`, `replace()`, `replaceAll()`, `split()`, and `search()`.

Let's look at each of them one by one.

The `search()` Method

The `search()` method searches for the match of a regular expression in a string and returns the index of the match.

```
const myStr =
  "fCC is the abbreviation for freeCodeCamp. freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp. Learn to code for free today.";
const re = /freeCodeCamp/;
```

```
const searchFCC = myStr.search(re);

console.log(searchFCC); //28
```

If the `search()` method finds no match, it returns `-1`:

```
const myStr =
  "fCC is the abbreviation for freeCodeCamp. freeCodeCamp doesn't charge you any money, that's
  why it's called freeCodeCamp. Learn to code for free today.";
const re = /FCC/;
const searchFCC = myStr.search(re);

console.log(searchFCC); //-1
```

You might be thinking using the `g` flag with the pattern would return the indices of all the matches, but this isn't the case. The `g` flag does not affect the `search()` method:

```
const myStr =
  "fCC is the abbreviation for freeCodeCamp. freeCodeCamp doesn't charge you any money, that's
  why it's called freeCodeCamp. Learn to code for free today.";
const re = /freeCodeCamp/g; //pattern with g flag
const searchFCC = myStr.search(re);

console.log(searchFCC); //28
```

If you want to get the indices of all the matches, you should use the `match()` or `matchAll()` method.

The `match()` Method

The `match()` method lets you specify a regex pattern as the parameter, then it runs through the string you use it against and returns an array containing the substring(s) that match the regex pattern.

```
const my_str = 'freeCodeCamp';
match = my_str.match(/free/);

console.log(match); // [ 'free', index: 0, input: 'freeCodeCamp', groups: undefined ]
```

You can also separate the regex pattern into a separate variable:

```
const my_str = 'freeCodeCamp';
const re = /free/;
const match = my_str.match(re);

console.log(match); // [ 'free', index: 0, input: 'freeCodeCamp', groups: undefined ]
```

If `match()` finds multiple matches, it returns all of them in the array, provided you use the `g` flag in the pattern:

```
const my_str =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp. Learn to
  code for free today.";
const re = /free/g;
const match = my_str.match(re);

console.log(match); // ['free', 'free', 'free']
```

If you expand the array, this is what it looks like:

match-method-matches

Since the result is an array, you should probably use `console.table()` instead of `console.log()` so you can see the indices of the matches:

```
const my_str =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp. Learn to
  code for free today.";
const re = /free/g;
const match = my_str.match(re);

console.table(match);
```

match-console-table

If the `match()` method finds no match, it returns `null`:

```
const my_str = 'freeCodeCamp';
const re = /ref/;
const match = my_str.match(re);

console.log(match); // null
```


The `matchAll()` Method

`matchAll()` is a hybrid of the `match()` method. It returns an iterator of all the substrings that match the regular expressions you provide. This means you have to use it with the `global` (`g`) flag.

Because it returns the iterator of all matches, `matchAll()` is a great option for looping through the matches of regular expressions.

An alternative to iterating through the matches of a regular expression is using the `exec()` method and `g` flag, then looping with a `while` loop this way:

```
const my_str =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp. Learn to
  code for free today.";
const re = /free/g;

let match;
while ((match = re.exec(my_str))) {
  console.log(match[0]); //

}

// free
// free
// free
```

With the `matchAll()` method, you don't need the `exec()` and `while` loop. All you need is a `for...of` loop to get the matches:

```
const my_str =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp. Learn to
  code for free today.";
const re = /free/g;
const matches = my_str.matchAll(re);

console.log(matches); // RegExpStringIterator {}

//loop through the matches with a for...of loop
for (const match of matches) {
  console.log(match);
}
```

This returns each `match`, their index, the test string, the length, and groups in their respective arrays:

matchAll-console

You can modify the console log to get only the matches and their index this way:

```
const my_str =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp. Learn to
  code for free today.";
const re = /free/g;
const matches = my_str.matchAll(re);

console.log(matches); // RegExpStringIterator {}

//loop through the matches with a for...of loop
for (const match of matches) {
  console.log(`Found a match ${match[0]} at index ${match.index}`);
}

/*
Output:
Found a match free at index 0
Found a match free at index 66
Found a match free at index 98
*/
```

You can also use the `Array.from()` method to do the same thing:

```
const my_str =
  "freeCodeCamp doesn't charge you any money, that's why it's called freeCodeCamp. Learn to
  code for free today.";
const re = /free/g;

Array.from(my_str.matchAll(re), (match) =>
  console.log(`Found a match ${match[0]} at index ${match.index}`)
);

/*
Output:
Found a match free at index 0
*/
```

```
Found a match free at index 66
Found a match free at index 98
*/
```

If the `matchAll()` method finds no match, it returns an empty iterator. And if you decide to loop through that empty iterator, there'll be nothing to see in the console.

The `replace()` Method

The `replace()` method does what its name implies. It searches for matches of a specified string or regular expression in a string and replaces them with the specified replacement string. It returns a new string with the replacements applied.

The `replace()` method is not as straightforward as `match()` and `matchAll()` because it accepts two parameters – a regular expression and the replacement string. Any substring of the test string that matches the regular expressions is then replaced with the replacement string.

If the regular expression does not include the global (`g`) flag, only the first match is replaced:

```
const myStr =
  'Elephants are very large animals. They are large to the extent that they can uproot a large tree.';
const re = /large/;
const replaceLarge = myStr.replace(re, 'massive');

console.log(replaceLarge); // Elephants are very massive animals. They are large to the extent that they can uproot a large tree.
```

If you use the `g` flag in the pattern, all the matches are replaced:

```
const myStr =
  'Elephants are very large animals. They are large to the extent that they can uproot a large tree.';
const re = /large/g;
const replaceLarge = myStr.replace(re, 'massive');

console.log(replaceLarge); // Elephants are very massive animals. They are massive to the extent that they can uproot a massive tree.
```

The `replaceAll()` Method

The `replaceAll()` method is relatively new because it became available in ECMAScript 2021. It is a hybrid of `replace()`.

Both `replace()` and `replaceAll()` do the same thing by taking a regular expression and a replacement string as parameters, and replacing all matches with the specified replacement string.

But unlike `replace()` which will only replace the first match if you don't use the `g` flag, `replaceAll()` replaces all the matches by default:

```
const myStr =
  'Elephants are very large animals. They are large to the extent that they can uproot a large tree.';
const re = /large/g;
const replaceLarge = myStr.replaceAll(re, 'massive');

console.log(replaceLarge); // Elephants are very massive animals. They are massive to the extent that they can uproot a massive tree.
```

If you don't use the `g` flag with `replaceAll()`, it throws a `TypeError`:

```
const myStr =
  'Elephants are very large animals. They are large to the extent that they can uproot a large tree.';
const re = /large/;
const replaceLarge = myStr.replaceAll(re, 'massive');

console.log(replaceLarge); // Uncaught TypeError: String.prototype.replaceAll called with a non-global RegExp argument
//    at String.replaceAll (<anonymous>)
```

The `split()` Method

The `split()` method takes a string or regex and splits the string you use it against into an array based on the string or regex you pass into it. The `split()` method also takes an optional `limit` parameter, a positive number. When you specify the `limit`, the splitting stops at that limit.

Wherever the `split()` finds a match, it creates a new item in the array. Here's how it works:

```
const myStr = "Codes don't lie. You're the one doing something wrong.";
const re = /\s/; // "\s" means white space - spacebar, backspace, tab, ENTER.

const splitedStr = myStr.split(re);
console.log(splitedStr);

/*
```

Output:

```
[
  'Codes', 'don't',
  'lie.', 'You're',
  'the', 'one',
  'doing', 'something',
  'wrong.'
]
*/
```

Here's how to use the `split()` method with the `limit` parameter:

```
const myStr = "Codes don't lie. You're the one doing something wrong.";
const re = /\s/; // "\s" means white space - spacebar, backspace, tab, ENTER.

const splitedStr = myStr.split(re, 5); // 5 is the limit here
console.log(splitedStr);

/*
output: [ 'Codes', 'don't', 'lie.', 'You're', 'the' ]
*/
```

How to Match Literal Characters in JavaScript Regular Expressions

As I pointed out earlier, literal characters are texts or strings you will write patterns for as they are.

If you want to match the text `hello`, `/hello/` should be your pattern. You can then use the `i` flag with it to match both `hello` and `Hello`:

```
const testString = 'hello';
const re = /hello/;
const re2 = /hello/i;

console.log(re.test(testString)); // true
console.log(re2.test(testString)); // true
```

If you want to match `freeCodeCamp`, the pattern should be just that. You can also create a pattern that matches `freeCodeCamp` in any case:

```
const testString = 'freeCodeCamp';
const re = /freeCodeCamp/;
const re2 = /freeCodeCamp/i; // match freeCodeCamp in any case

console.log(re.test(testString)); // true
console.log(re2.test(testString)); // true
```

You can also match digits using literal characters:

```
const num = 10234;
const re = /2/;

console.log(re.test(num)); //true
```

How to Use Character Sets in JavaScript Regular Expressions

As a reminder, a character set is a group of characters enclosed in square brackets. They provide a way to specify a set of characters from which the regex engine can match a single character at a specific position in a test string.

Character sets allow you to specify a range of characters, individual characters, or a combination of both.

Here are common examples of popular character sets in regular expressions:

- `[abc]`: matches either `a`, `b`, or `c`
- `[aeiou]`: matches any vowel character
- `[a-z]`: matches any lowercase letter from `a` to `z`
- `[A-Z]`: matches any uppercase letter from `A` to `Z`
- `[0-9]`: matches any digit from 0 to 9

Let's look at how to match each of the above character sets in JavaScript regular expressions:

```
// uppercase character set
const hcaseRe = /[A-Z]/;
const hcaseStr = 'freeCodeCamp is cool';

console.log(hcaseRe.test(hcaseStr)); //true

// vowels character set
```

```
const vowelsRe = /[aeiou]/;
const vowelsStr = 'Imagine how pronunciation would have been without vowels';

console.log(vowelsRe.test(vowelsStr)); //true

// [abc] character set
const abcSetRe = /[abc]/;
const abcSetStr = 'freeCodeCamp is totally free';

console.log(abcSetRe.test(abcSetStr)); //true

// number character set
const numRe = /[0-9]/;
const numStr = 'Thank God for Arabic numerals 0 to 9.';

console.log(numRe.test(numStr)); //true
```

What are Metacharacters?

In regular expressions, metacharacters are characters that have special meanings beyond their literal meaning.

Metacharacters are the backbone of regular expressions. They serve as the building blocks for constructing better regex patterns and defining the behavior of the regular expression engine you're using, but with an extra learning curve.

This part of the book is where you will learn about topics such as:

- Anchors
- Word boundaries
- How to specify character ranges
- How to match every occurrence with the wildcard
- Alternation
- Greediness and laziness of regular expressions and how to prevent greediness

And lots more.

If you want to match any metacharacter as a literal character, you have to escape it with a backslash (`\`). And if there's a metacharacter represented by a word, you have to escape it with the backslash too. So, the backslash is also a separate metacharacter.

There's a metacharacter to negate most metacharacters. For instance, `\b` and `\s` represent the word boundary and space metacharacters. If you want to negate them, you can use `\B` and `\S`

respectively. That's the pattern most metacharacters follow – the small letter is the metacharacter and the capital letter negates it.

Metacharacters are categorized into single and double metacharacters. As the name implies, single metacharacters have a "single" character and double metacharacters have a "double" character.

Most metacharacters are also called shorthand character classes. As we look at each metacharacter, you will see whether it is a single or double metacharacter.

The Word and Non-word Metacharacters

Represented by `\w`, the word metacharacter is a shorthand character class that matches all word characters. Word characters are alphanumeric characters and underscores. So, they are `a-z`, `A-Z`, `0-9`, and underscore (`_`).

Here's what happens when you use `\w` in a regex tester:

w-matches

And here's how it works in JavaScript:

```
const testStr =  
  'Every alphanumeric character (a to z and 0 to 9) and underscore (_) is a word character';  
const wordCharacterRe = /\w/g;  
  
console.log(testStr.match(wordCharacterRe));
```

Since word characters are alphanumeric characters and underscores, you can simulate the `\w` metacharacter by putting all the examples in a character set:

```
const testStr =  
  'Every alphanumeric character (a to z and 0 to 9) and underscore (_) is a word character';  
const wordCharacterRe = /[a-z A-Z 0-9_]/g;  
  
console.log(testStr.match(wordCharacterRe));
```

The non-word metacharacter is the opposite of the word metacharacter and it is represented by an escaped capital letter W (`\W`).

The non-word metacharacter matches every other character apart from alphanumeric characters and the underscore. That includes spaces, punctuation marks, and symbols:

w-matches-1

Here it is in action in some JavaScript code:

```
const testStr =  
  'Every character apart from alphanumeric characters (a to z and 0 to 9) and underscore (_)  
  is a non-word character';  
const nonWordCharacterRe = /\W/g;  
  
console.log(testStr.match(nonWordCharacterRe));
```

Since you can represent the word metacharacter by putting all the characters in a character set, you may be wondering how you can do the same for the non-word metacharacter.

That's where the negated character set comes in. The caret (^) is used for negation. It is one of the two **anchor metacharacters**, which we'll look at next.

The Anchor Metacharacters

Caret (^) and dollar sign (\$) are the two anchor metacharacters. They are both single metacharacters.

The caret anchors the regex pattern to the start of a line or string, so you can call it a "start of line anchor".

For example, if you want to match the text "freeCodeCamp" and you want to make sure it's at the start of the line or a string, you can use the caret this way:

fcc-anchor-match

If the `freeCodeCamp` text is not at the start of the line, there won't be a match:

fcc-anchor-no-match

Here are the two cases in JavaScript code:

```
const testStr =  
  "freeCodeCamp doesn't charge you any money. That's why it's called freeCodeCamp because.  
  Learn to code for free today."; // has "freeCodeCamp" at the start of the line  
  
const testStr2 =  
  "It's called freeCodeCamp because freeCodeCamp doesn't charge you any money. Learn to code  
  for free today."; // does not have "freeCodeCamp" at the start of the line  
  
const startAnchorRe = /^freeCodeCamp/;
```

```
console.log(startAnchorRe.test(testStr)); //true
console.log(startAnchorRe.test(testStr2)); //false
```

The dollar sign metacharacter is the opposite of the caret. It anchors the regex pattern to the end of the line or string. So, there will only be a match if the target text is at the end of the line.

To use the `$` metacharacter, it has to be the last character in your pattern:

dollar-meta-match

If the target string has more than one line and the target text is at the end of each line, the last one matches:

dollar-last-match

To correct this behavior, you have to use both the `g` and `m` flags:

dollar-multiple-match

Here are all the cases in JavaScript code:

```
const testStr =
  "The lion is not the king of the jungle because of its strength, the lion is the king of the
  jungle because it's never intimidated";

const testStr2 = `The lion is not the king of the jungle because of its strength, the lion is
the king of the jungle because it's never intimidated

This is another line that ends with intimidated

And this is the last line that ends with intimidated

And this is the last line that ends with intimidated`;

const re = /intimidated$/;
const re2 = /intimidated$/gm;

console.log(re.test(testStr)); // true
console.log(re.test(testStr2)); // true
console.log(re2.test(testStr2)); // true
```

If the target text is not at the end of the line, there won't be any match:

```
const testStr =  
  "A lion can never be intimidated because it's the king of the jungle";  
const re = /intimidated$/;  
  
console.log(re.test(testStr)); // false
```

When you use both the dollar and caret metacharacters with the `g` and `m` flags, they don't just match at the start and end of a line, they find the matches at the start and end of each line:

```
//dollar with g and m flags  
const testStr1 = `The lion is not the king of the jungle because of its strength, the lion is  
the king of the jungle because it's never intimidated  
  
Another line with intimidated  
  
And another line with intimidated`;  
  
const re1 = /intimidated$/gm;  
const matches1 = testStr1.match(re1);  
  
console.log(matches1); // [ 'intimidated', 'intimidated', 'intimidated' ]  
  
// caret with g and m flags  
const testStr = `freeCodeCamp doesn't charge you any money. That's why it's called  
freeCodeCamp because. Learn to code for free today.  
  
freeCodeCamp starts this line  
  
freeCodeCamp starts this line too  
`;  
  
const re2 = /^freeCodeCamp/gm;  
const matches2 = testStr.match(re2);  
  
console.log(matches2); // [ 'freeCodeCamp', 'freeCodeCamp', 'freeCodeCamp' ]
```

As I pointed out earlier, the caret metacharacter is typically used for negating a character set or any other character. With that, you tell the regex engine in use not to match that character or each of the character sets.

For example, if you have the pattern `[^a]`, then all letters "a" in the test string won't be returned as matches:

unmatch-As

If you have the pattern `[^aeiou]`, all the vowels in the test string won't be returned as matches:

unmatch-vowels

If you have the pattern `[^a-zA-Z0-9_]`, that's equivalent to the non-word metacharacter (`\W`):

non-word-char-class

The Digit and Non-digit Metacharacters

The digit metacharacter is represented by `\d`. You can negate it with `\D`, so `\D` is the non-digit metacharacter.

`\d` matches all numbers (0 to 9), so it is a shorthand character class for `[0-9]`. So, if you have a string and you want to extract the numbers from it, you can use the `\d` metacharacter. But you have to use it with the `g` flag so it matches every number in the test string:

Screenshot-2023-07-25-at-12.27.21

You can use the `match()` method to extract the numbers in JavaScript too:

```
const testStr =
  'Arabic numerals are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. From those ten numbers, you can write
  any number you want, including nonillion and decillion.';

const re = /\d/g;

console.log(testStr.match(re));

/* output
[
  '0', '1', '2', '3',
  '4', '5', '6', '7',
  '8', '9'
]
*/
```

A more straightforward example is matching dates since dates are mostly in numbers. For example, if you want to match a date in the format `dd/mm/yyyy`, you can match it with the pattern

```
/\d\d\/\d\d\/\d\d\d\d/:
```

```
const date = '22/04/2023';  
const re = /\d\d\/\d\d\/\d\d\d\d/;  
  
console.log(re.test(date)); // true
```

Since you can also have a period or hyphen as the separator of a date, you can account for those too by putting all the possible separators in a character set:

```
const slashSeparatedDate = '22/04/2023';  
const hyphenSeparatedDate = '22-04-2023';  
const periodSeparatedDate = '22.04.2023';  
  
const re = /\d\d[/.-]\d\d[/.-]\d\d\d\d/;  
  
console.log(re.test(slashSeparatedDate)); // true  
console.log(re.test(hyphenSeparatedDate)); // true  
console.log(re.test(periodSeparatedDate)); // true
```

N.B.: The pattern above matches a date but also an invalid date like `99/45/2022`. A better way to match dates is provided in the applications of the regex chapter.

Another example is matching phone numbers. For example, US phone numbers are in the format `(123) 456-7890`. You can use the pattern `/\(\d\d\d\) \d\d\d-\d\d\d\d/`:

```
const USPhone = '(123) 456-7890';  
const re = /\(\d\d\d\) \d\d\d-\d\d\d\d/;  
  
console.log(re.test(USPhone)); // true
```

The non-digit metacharacter is the opposite of the digit metacharacter. It matches all non-digit characters. That is, alphabets, spaces, and symbols. In other words, it is the shorthand character class for `[^0-9]`.

If you want to extract all non-digit characters in a string, you can use the `\D` metacharacter: `\D` matches

This is it in JavaScript code:

```
const testStr =  
  'Arabic numerals are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. From those ten numbers, you can write
```

```
any number you want, including nonillion and decillion.';

const re = /\D/g;

console.log(testStr.match(re));

/* output
A total of 137 matches is too much to show here, but you can test it out yourself.
*/
```

The Square Brackets Metacharacter

You've already seen the square brackets (`[]`) metacharacter in action. Square brackets are used for specifying a character class, or character set. And if you want to match them as a literal character, then you have to escape them.

One thing to have in mind is that some metacharacters lose their meanings inside the character set. The exceptions to this are:

- The caret (`^`) which you can use to negate a character set
- The hyphen (`-`) which you can use to specify ranges

N.B.: Sometimes, you might encounter a situation where you have to escape some metacharacters inside a character set.

If you want to match any of those characters in a character set, you have to escape it. If you are just passing the three of those characters in directly, you don't need to escape them if the caret is not the first character.

```
const testStr =
  'If you want to match the caret (^), hyphen and (-) symbols in a character set, you might
  not have to escape them.';

const re = /[-^]/g;

console.log(testStr.match(re)); // [ '^', '-' ]
```

But if the caret is the first character in the character set alongside some word and non-word character, you should escape it, otherwise it will negate all other characters:

unescaped-caret

The Word Boundary and Non-word Boundary Metacharacters

The word boundary metacharacter is represented by `\b` and the non-word boundary metacharacter is represented by `\B`. Both let you match a specific part of a string where a word character and a non-word character exist.

Word boundary (`\b`) matches a position between a word character (`\w`) and a non-word character (`\W`), and vice versa. It can be useful when you want to match a certain word in a string, or if you want to make sure a particular word or character is in a string.

Here's an example in a regex tester:

b-match

And the same example in JavaScript code:

```
const myStr =
  'A Tiger can do everything a lion does, apart from being a family man.';
const re = /\blion\b/;

console.log(myStr.match(re));

/*
Output:
[
  'lion',
  index: 28,
  input: 'A Tiger can do everything a lion does, apart from being a family man.',
  groups: undefined
]
*/
```

If you use a `g` flag with the pattern and use the `match()` method, all the matches will be returned – as expected:

```
const myStr =
  'A Tiger can do everything a lion does, apart from being a family man. Not even a tiger can
  intimidate a lion within his family.';
const re = /\blion\b/g;

console.log(myStr.match(re)); // [ 'lion', 'lion' ]
```

On the other hand, the non-word boundary (`\B`) is the opposite of the word boundary (`\b`). So, it matches everywhere a word boundary won't return a match. For example, "thin" in "everything":

thing-everything

And also "code" in "freeCodeCamp" when you use the case insensitive (`i`) flag:

code-freeCodeCamp

You can see that the first "code" in the text wasn't the match returned. That's the power of word and non-word boundary metacharacters.

Here's what the two reveal in JavaScript code:

```
const myStr1 =
  'A Tiger can do everything a lion does, apart from being a family man.';
const myStr2 = 'Learn to code for free on freeCodeCamp.';

const re1 = /\Bthin\b/;
const re2 = /\Bcode\b/i;

console.log(myStr1.match(re1));
console.log(myStr2.match(re2));

/*
Output:
[
  'thin',
  index: 20,
  input: 'A Tiger can do everything a lion does, apart from being a family man.',
  groups: undefined
]
[
  'Code',
  index: 30,
  input: 'Learn to code for free on freeCodeCamp.',
  groups: undefined
]
*/
```

The Parenthesis Metacharacter

The parenthesis metacharacters (`(` and `)`) let you create grouping and capturing. With them, you can treat any group of characters as a single unit and apply a common modifier or quantifier to them.

Parenthesis is also used for creating both lookahead and lookbehind assertions.

When you create the group and assertions, you can reference them later in the same pattern with a backslash and the order in which they appear. For example, you can reference the first group by specifying `\1` in the pattern.

In this book, a whole chapter is dedicated to grouping and capturing. There, you will learn more about grouping and capturing so you can see the parenthesis metacharacters in action.

The Space and Non-space Metacharacters

It is impossible for text to make sense without spaces. Not just a "space", but also other space characters like tabs, carriage returns, and new lines. This is why the space and non-space metacharacters are made available in regular expressions.

The space metacharacter is represented by `\s` and the non-space metacharacter is represented by `\S`.

`\s` matches all space characters:

s-match

And `\S` matches all non-space metacharacters:

S-matches

Here's how both the `\s` and `\S` metacharacters work in JavaScript code:

```
const myStr = 'Learn to code for free on freeCodeCamp';
const spaceRe = /\s/g;
const nonSpaceRe = /\S/g;

console.log(myStr.match(spaceRe)); // [' ', ' ', ' ', ' ', ' ', ' ', ' '];

console.log(myStr.match(nonSpaceRe));
// [
// 'L', 'e', 'a', 'r', 'n', 't',
// 'o', 'c', 'o', 'd', 'e', 'f',
// 'o', 'r', 'f', 'r', 'e', 'e',
// 'o', 'n', 'f', 'r', 'e', 'e',
```

```
// 'C', 'o', 'd', 'e', 'C', 'a',  
// 'm', 'p'  
// ]
```

One cool thing you can do with `\s` in JavaScript is to replace all spaces with say, a hyphen, or any other thing you want:

```
const myStr = 'Learn to code for free on freeCodeCamp';  
const replaceHyphen = myStr.replace(spaceRe, '-');  
  
console.log(replaceHyphen); // Learn-to-code-for-free-on-freeCodeCamp
```

The space metacharacter does not just match the spacebar you press on the keyboard of your device. It also matches:

- A tab character
- A carriage return character
- A new line character
- A vertical tab character
- And a form feed character

Here's an example:

s-match-all

You can't see the match for the carriage return but it's there:

s-match-all-view-1

If you want to match each of those space characters, they also have their unique metacharacters:

- `\t` for tab
- `\r` for carriage return
- `\n` for new line
- `\v` for vertical tab
- `\f` for form feed.

You should be aware that most of the time, `\s` is all you need because it can do the matching for any space character.

The Pipe Metacharacter

Also known as the `OR` operator, the pipe metacharacter is represented by the pipe symbol (`|`). It lets you specify multiple alternatives for matching.

The pipe matches the character preceding it, or the character that follows it. For example, if you have `website|web\sapp` as your pattern, then one or both of `website` and `web app` will be returned as the match:

web-webapp-alt

The evaluation goes from left to right. If a match is found on the left, it returns the match. And if there's no match on the left, the character on the right-hand side is evaluated for a possible match. If both characters on the left and right are in the test string, then both are returned as matches.

You can also have more than two characters separated by the pipe symbols. For instance, the pattern `/o|a|i|re/` would match `o`, `a`, `i`, and `re`:

a-o-i-alt-match

There's no limit to the characters you can separate with it.

You can see I used the `g` flag in those examples. If you don't use the `g` flag and both the left and right characters are matches, only the first match in the test string will be returned:

first-occur-i-o-a

Here's a clearer example:

clearer-alt

Here's how using the `OR` operator works with the `g` flag in code:

```
const myStr = 'The website and web app are running fine';
const re = /website|web\sapp/g;

console.log(myStr.match(re)); // returns [ 'website', 'web app' ] because of the g flag
```

And here's how it works without the `g` flag:

```
const myStr = 'The website and web app are running fine';
const re = /website|web\sapp/;

const matches = myStr.match(re);

for (const match of matches) {
  console.log(match); // returns "website" and ignores web app because there's no g flag
}
```

How to Match Repeated Characters With Quantifiers

Repeated characters occur when the same character exists in multiple numbers consecutively.

When you have a repeated character in your test string, you don't need to repeat a particular character in your pattern to match it. That's because there are metacharacters available for **one or more matches**, **zero or more matches**, and **zero or one matches**, AKA **optional matches**.

One or More Matches with the Addition Sign Metacharacter

As you can guess, the addition sign metacharacter is represented with a plus (+). You can also call it the "one or more quantifier".

If you want a particular character to be repeated one or many times, that's what the addition sign metacharacter does.

For example, the pattern, `/fe+d/` will match any word with one letter `e` or multiple letters `e` that occur consecutively. For instance, `fed` and `feed`:

one-or-more-e

A practical example in JavaScript is extracting vowels in a test string while limiting occurrences by making sure multiple vowels that follow one another are also returned:

```
const myStr = 'You should plant trees to save mother earth';
const re = /[aeiou]+/gi;

console.log(myStr.match(re));

/*
Output:
[
  'ou', 'ou', 'a',
  'ee', 'o', 'a',
  'e', 'o', 'e',
  'ea'
]
*/
```

You can also append the addition sign metacharacter to other metacharacters. For example, `/d+/` would match one or more digits:

d--matches

You can also add the `+` metacharacter to a character set to repeat it one or more times. In the screenshot below, the pattern `/f[a-z]+/` would match one or more letter `f` followed by any set of small letters:

f-set-zero-or-more

Zero or More Matches with the Asterisk Metacharacter

The asterisk metacharacter (`*`) matches zero or many occurrences of the character it comes after. You can also call it a "zero or more quantifier".

So, if you want a character to be repeated zero or more than one time, you can use the asterisk metacharacter. A basic example is using the pattern `/go*d/` would match any word that starts with the letter `g` followed by any number of the letter `o`, and ending with the letter `d`:

gd-zero-or-more

Just like you can do with the plus metacharacter, you can also append the asterisk metacharacter to any other metacharacter. For example, you can match empty strings with the pattern `/\s*/`:

match-empty-string

Doubting that? Here it is in JavaScript code:

```
const re = /\s*/;
const emptyString = '';

console.log(re.test(emptyString)); // true
```

I didn't know matching empty strings was as straightforward as this until I got to this point in the book!

Again, like the plus metacharacter, you can also add the `*` metacharacter to a character set to repeat it zero or more times:

f-set-one-or-more

Here's the same thing in JavaScript code:

```
const myStr = 'You can make yourself free from diseases';
const re = /f[a-z]*/g;

console.log(myStr.match(re)); // [ 'f', 'free', 'from' ]
```

You can see the `f` in the word `yourself` is even a match too. That's one way to deduce that the asterisk (`*`) returns more matches than the addition sign (`+`) metacharacter because it is greedier. You will learn about greediness of a regular expression in the closing part of this chapter.

Zero or One Matches with the Question Mark Metacharacter

The question mark metacharacter (`?`) is also known as the zero or one quantifier. It lets you make the character that precedes it optional, so it plays an important role in preventing greediness.

For example, the pattern `/ab?c/` will match `abc` and `ac`, but never `abbbc` or any other numbers of `b` between the `a` and `c`:

abc-optional

This is not the case with the other two metacharacters for matching repeated characters (`+` and `*`). The pattern `/ab*c/` will match all of `abc`, `ac`, `abbbc`, and `abbbbbbbc` while `/ab+c/` will leave out `ac`:

```
const myStr = 'abc ac abbbc abbbbbbbc';
const re1 = /ab*c/g;
const re2 = /ab+c/g;
const re3 = /ab?c/g;

console.log(myStr.match(re1)); // [ 'abc', 'ac', 'abbbc', 'abbbbbbbc' ]
console.log(myStr.match(re2)); // [ 'abc', 'abbbc', 'abbbbbbbc' ]
console.log(myStr.match(re3)); // [ 'abc', 'ac' ]
```

A better example is tailoring a regex pattern to match words that have different spellings due to the small variations in British and American English. For example, `color` and `colour`:

colou-r-optional

There's also `centre` and `center`:

cente-re--optional

You can extract those words in JavaScript. You can't use the `match()` method for that because it causes some unexpected behaviors when used with the `?` metacharacter.

Here's how I was able to do it for `color` and `colour`:

```
const myStr = 'The words center and centre are homophones';
const re = /cente?re?/g;

let match;
```

```
const matches = [];  
  
while ((match = re.exec(myStr)) !== null) {  
    matches.push(match[0]);  
}  
  
console.log(matches); // ["center", "centre"]
```

I used the same approach to extract `center` and `centre`:

```
const myStr =  
    'It is "colour" in British English and "color" in American English';  
const re = /colou?r/g;  
  
let match;  
const matches = [];  
  
while ((match = re.exec(myStr)) !== null) {  
    matches.push(match[0]);  
}  
  
console.log(matches); // [ 'colour', 'color' ]
```

Many times, it's challenging knowing which to use for character repetition between these three metacharacters – `*`, `+`, and `?`. It can even be hard to get used to what each of them does if you're just starting out with regular expressions.

Be aware that identifying them and knowing which to use between them is not a herculean task. Here are some things to note about the three of them:

- Asterisk (`*`) means "zero or many": use it if you want a character not to appear in the target string or you want the same character to be more than one
- Plus (`+`) means "one or many": use it if you want a character to appear once or more than once in the target string
- Question mark (`?`) means "zero or one": use it if you want a character to be optional in the target string.

How to Specify Match Quantity with the Curly Braces Metacharacter

Quantifiers let you indicate the quantity or frequency of a preceding character in a pattern with curly braces (`{ }`). With those braces, you can specify an exact quantifier, a minimum quantifier, and a range quantifier.

The Range Quantifier

The general syntax for the range quantifier looks like this:

```
char{n1,n2}
```

- `cha` is any character you're applying the quantifier to
- `n1` is the minimum number of times you want the character to repeat
- `n2` is the maximum number of times you want the character to repeat

An example is the pattern `/a{3,6}/`. This means you want to match between three and six letters `a`:

a-3-6--match

If you have more than six letters `a` in the test string, the first six will match:

a-range-error

To fix this, you can surround the pattern in a word boundary:

a-range-error-fix

You can also attach the range quantifier to metacharacters. For example, you can extract any number that is at least in hundreds this way:

```
const myStr =  
  'The marathon had 500 participants, with 251 finishing under 3 hours, and the winner crossed  
  the line at 4800 seconds.';  
const re = /\b\d{3,6}\b/g;  
  
console.log(myStr.match(re)); // [ '500', '251', '4800' ]
```

The Minimum Quantifier

The minimum quantifier lets you specify the minimum number of times you want the character that precedes it to match. You can do this by putting a comma right after the number in the curly brace. The general syntax looks like this: `{n,}`.

For example, the pattern `/a{3,}/` means you want a minimum of three letters `a`. In this case, one letter `a` and two letters `aa` won't be a match, but three letters `aaa` and upward would be returned as matches:

a-3---match

Let's extract those matches with the `match()` method:

```
const myStr =  
  '"a" won\'t match here. "aa" won\'t match too, but "aaa" is a match, "aaaa" is also a match,  
  and every other number of "a"';  
const re = /a{3,}/g;  
  
console.log(myStr.match(re)); // [ 'aaa', 'aaaa' ]
```

The Exact Quantifier

The exact specifier is represented by `{n}`. In this case, `n` stands for the exact number of times you want that character to be repeated. For instance, the pattern, `/a{3}/` means you want `a` to be repeated three times

exact-a-3-

Unfortunately, a match is returned anywhere there are three letters `a` that follow one another. You can prevent this behavior with word boundary (`\b`):

exact-a-3--fixed-

That way, you can extract the abbreviations, `AAA` from a string using the `match()` method. Below is an example:

```
const myStr =  
  "There is American automobile association (AAA)and there is Australian automobile  
  association (AAA). What I've never seen is AAAA or AAAAAA.";  
const re = /\ba{3}\b/gi;  
  
console.log(myStr.match(re)); // [ 'AAA', 'AAA' ]
```

Remember the pattern I wrote to match dates in the `dd/mm/yyyy` format? You can make it better and easier to read with the exact quantifier like this:

```
\d{2}[/.-]\d{2}[/.-]\d{4}
```

Everything still works fine:

```
const slashSeparatedDate = '22/04/2023';  
const hyphenSeparatedDate = '22-04-2023';  
const periodSeparatedDate = '22.04.2023';
```

```
const re = /\d{2}[/.-]\d{2}[/.-]\d{4}/;

console.log(re.test(slashSeparatedSate)); // true
console.log(re.test(hyphenSeparatedDate)); // true
console.log(re.test(periodSeparatedDate)); // true
```

You can also make the pattern that matches the US phone number better and shorter with the same approach:

```
\(\d{3}\) \d{3}-\d{4}
```

Everything still works fine too:

```
const USPhone = '(123) 456-7890';
const re = /\(\d{3}\) \d{3}-\d{4}/;

console.log(re.test(USPhone)); // true
```

The Wildcard Metacharacter

The wildcard metacharacter is represented by a dot (`.`), so you can also call it the dot metacharacter.

The wildcard lets you match any character apart from a new line (`\n`). That means you can use it to match alphanumeric characters, spaces, and symbols.

wcard-match

You can also attach the wildcard metacharacter to another metacharacter. For example, the pattern `/\d./g` should match at least a number and everything that follows it:

digit-greedy

You can see that the pattern is transcending beyond the digits by matching the spaces after them. This is what is called **greediness**.

The pattern, `/\d.*g` is even more greedy because it will match everything after it encounters the first number:

super-digit-greedy

It's the same in code:

```
const myStr =
  'An example of a two-digit number is 20. 100 is a three-digit number. 300 and 900 are also
  three-digit numbers.';
const re = /\d.*/g;

console.log(myStr.match(re)); // [ '20. 100 is a three-digit number. 300 and 900 are also
  three-digit numbers.'
```

If you want the wildcard to match a new line too, you can use the `s` flag. Here's an example:

```
let codeBlock = `
  function add(x, y) {
    /* This is a function
    that takes two numbers
    and adds them together. */
    return x + y;
  }
`;

let commentRegex = /\/*\*(.*)\*\/s; // gets everything between /* and */

const match = codeBlock.match(commentRegex);
console.log(match);
```

Here's the result:

dotAllRes-1

You can use the `dotAll` property to check if the `s` flag is used in the pattern:

```
let codeBlock = `
  function add(x, y) {
    /* This is a function
    that takes two numbers
    and adds them together. */
    return x + y;
  }
`;

let commentRegex = /\/*\*(.*)\*\/s; // gets everything between /* and */
const match = codeBlock.match(commentRegex);
```

```
console.log(commentRegex.dotAll) // true;
```

You can extract the match with an if statement:

```
let codeBlock = `
  function add(x, y) {
    /* This is a function
      that takes two numbers
      and adds them together. */
    return x + y;
  }
`;

let commentRegex = /\/*(.*)\*/s; // gets everything between /* and */

const match = codeBlock.match(commentRegex);

if (match) {
  console.log(match[1]);
}

/*
Output:
This is a function
    that takes two numbers
    and adds them together.
*/
```

Because the wildcard always matches any character it encounters apart from a new line, it is better not to use it unless it is absolutely necessary. For every character the wildcard matches, there is always another way to match it.

Greediness and Laziness in Regular Expressions

By default, regular expression patterns are greedy, meaning they always try to match as many as possible characters. But the concept of greediness is primarily applicable to quantifiers (`*`, `+`, `?`, and `{}`) and the wildcard (`.`).

For Example, the pattern `/f.*h/gi` will match as many characters as possible after encountering an `f` in the target string:

asterisk-greedy

Same for the pattern, `/f.*h/gi`:

plus-greedy

It's the same in code:

```
const myStr = 'The fresh fish was caught in the Finnish lake';
const re = /f.*h/gi;

console.log(myStr.match(re)); // [ 'fresh fish was caught in the Finnish' ]
```

Laziness is the opposite of greediness and it's the way you stop greediness. On many occasions, if you want to stop greediness, all you need is to apply the **zero or ones quantifier** `(?)` to the metacharacter causing the greediness.

Here's how I stopped the greediness of the asterisk metacharacter:

make-asterisk-lazy

I stopped it for the plus metacharacter the same way:

make-plus-lazy

I can now safely extract every word that starts with `f` and ends with `h`:

```
const myStr = 'The fresh fish was caught in the Finnish lake';
const re = /f.*?h/gi;

console.log(myStr.match(re)); // [ 'fresh', 'fish', 'Finnish' ]
```

Chapter 6: Grouping and Capturing in Regex

What is Grouping?

Grouping means treating a regex pattern or a part of a regex pattern as a single unit. To achieve grouping, you surround the pattern or the part of the pattern you want to group in parenthesis `()`

and `)`).

After you've grouped the part of the pattern you want to, you can then refer back to it through a process we call "backreferencing" in regular expressions.

The groups you define in a pattern refer to the target string or text and not the pattern itself. You'll see this in action when it's time to discuss backreferencing.

After grouping, you can then apply a quantifier to that group since all the patterns in it are a unit.

Let's say you have a group of the ids `z8g4g4 ga1v4g f4k7f9 bb3b2b d6b4t5 d4cm3d e9f5y6 ggj64 mgtyqg m0foh9` and you want to find out which of them follow the pattern `letter number letter number letter number`. The pattern `[a-z]\d[a-z]\d[a-z]\d` can do that for you:

without-grouping

Using grouping, you can make the pattern shorter by grouping the `[a-z]\d` sequence and applying an exact quantifier of `3` to it:

```
([a-z]\d){3}
```

with-grouping

When you use grouping in a pattern, especially if you have multiple groups in the same pattern, you can use the `exec()` method to extract each of the groups.

A good example to illustrate this is a date in any acceptable format, for example `dd/mm/yyyy`.

Here's how I group the pattern `\d\d[/.-]\d\d[/.-]\d\d\d\d` into `dd`, `mm`, and `yyyy`:

```
(\d\d)[/.-](\d\d)[/.-](\d\d\d\d)
```

I used the `exec()` method this way:

```
const re = /(\d\d)[/.-](\d\d)[/.-](\d\d\d\d)/;  
const date = '22-03-2023';  
  
const execRes = re.exec(date);  
console.log(execRes);
```

This is what the result looks like in the console:

exec-group-res

In the array, you can see that:

- there is the whole date in the index `0`
- the index `1` has the `day`
- the index `2` has the month`
- and the index `3` has the `year`

You can then use array referencing to get all of those figures:

```
const re = /(\d\d)[/.-](\d\d)[/.-](\d\d\d\d)/;
const date = '22-03-2023';

const execRes = re.exec(date);

console.log(`The full date is ${execRes[0]}`); // The full date is 22-03-2023
console.log(`The day is ${execRes[1]}`); // The day is 22
console.log(`The month is ${execRes[2]}`); // The month is 03
console.log(`The year is ${execRes[3]}`); // The year is 2023
```

You can also use this approach to extract a username and domain from an email:

```
function extractUsernameAndDomain(email) {
  const re = /([a-z]{2,})@([a-z]{3,}\.com)/;
  const result = re.exec(email);

  console.log(`The username is ${result[1]}`);
  console.log(`The domain is ${result[2]}`);
  console.log(`The full email is ${result[0]}`);
}

extractUsernameAndDomain('janedoe@gmail.com');

/*
Output:
The username is janedoe
The domain is gmail.com
The full email is janedoe@gmail.com
*/
```

This behavior of grouping in which each match of the pattern is separated in an array according to the groups is the reason groups are also called "capturing" groups. This way, you don't need the `split()` method of JavaScript or any other programming hacks to get each of the groups on those dates.

How to Reference Captured Groups with Backreferences

Since groups are captured by default, you can refer back to them. To do this, you use a backslash (`\`) and then the order of the group in the pattern. For example, you can reference the first group with `\1` and the third group with `\3`. No zero indentation.

Let's say you want to match "tsetse" fly in the text `There are many tsetse flies in the tropics`. If you group the text "tse" first and use the `g` flag, you'll get two matches:

tse-group-warn

You can refer back to that `tse` group with `\1` and you'll have a single match:

tse-group-right

It's very important to note that when you use a capturing group, the grouping refers to the target string (or text) and not the pattern itself. The reason why the pattern `/(tse)\1/` returns a match in the last example is because of the "tse" in the text and not the "tse" in the pattern.

To illustrate this, let's use a date again, since the month or date and the separators can repeat and can be different. I will use the pattern `(\d\d)([/.-])\1\2(\d\d\d\d)` for matching dates that I grouped in one of the previous examples. Remember the pattern successfully matches a date:

date-match-group

I can group the separator too and refer back to it for the second separator. I can also refer back to the day part of the date to match the month, since they both look for two digits.

Here's the new pattern now:

```
(\d\d)([/.-])\1\2(\d\d\d\d)
```

I can make the pattern shorter with an exact quantifier:

```
(\d{2})([/.-])\1\2(\d{4})
```

The new pattern successfully matches the same date:

date-match-group-same

But the reason there's a match in the example above is that the separators are the same and the day and month are the same.

If the day is different from the month, there won't be a match:

no-match-because-of-day-month-difference

If the separators are different too, there also won't be a match:

no-match-because-of-separator-difference

But remember that if both are the same, there will be a match:

date-match-group-same-1

That is the reason why the groups in a pattern refer to the target string (or text) and not the pattern itself.

It is also possible to make a group non-capturing. That way, you won't be able to refer to it in the pattern. To create a **non-capturing** group, you use a question mark and a colon right after the opening parenthesis.

The syntax for that looks like this:

```
(?: chars)
```

no-match-non-capture

Because of this, the text does not match the pattern anymore. To make it match again I have to:

- remove the first backreference (`\1`)
- define `\d{2}` for the date
- change the reference to the separator from `\2` to `\1`

Here's the new pattern:

```
(?:\d{2})([/.-])\d{2}\1(\d{4})
```

And now the date matches the pattern:

match-non-capture

How to Use the `d` Flag and `hasIndices` Property with Groups

The `d` flag adds index information to match objects for capture groups. This way, you won't just know what was matched by each capture group, but also where that match was found in the input string.

Let's look at how this works with the grouping for matching dates:

```
const re = /(\d\d)[/.-](\d\d)[/.-](\d\d\d\d)/d;
const date = '22-03-2023';

const match = re.exec(date);
console.log(match);
```

The result contains an array of objects detailing the total position of all matches, and the position of each match:

dFlagRes

If you want to see those indices, you can use `.indices` to see them:

```
const re = /(\d\d)[/.-](\d\d)[/.-](\d\d\d\d)/d;
const date = '22-03-2023';

const match = re.exec(date);

console.log(match.indices);
```

dFlagHasIndicesRes

You can also extract those indices separately:

```
const re = /(\d\d)[/.-](\d\d)[/.-](\d\d\d\d)/d;
const date = '22-03-2023';

const match = re.exec(date);

console.log(`The full index range is ${match.indices[0]}`); //The full index range is 0,10
console.log(`The day index range is ${match.indices[1]}`); // The day index range is 0,2
console.log(`The month index range is ${match.indices[2]}`); // The month index range is 3,5
console.log(`The year index range is ${match.indices[3]}`); // The year index range is 6,10
```

And finally, you can check if the `d` flag is really used with the `hasIndices` property:

```
const re = /(\d\d)[/.-](\d\d)[/.-](\d\d\d\d)/d;
const date = '22-03-2023';

console.log(re.hasIndices); // true
```

Chapter 7: Lookaround Groups: Lookaheads and Lookbehinds

What are Lookaround Groups?

Lookaround assertions are non-capturing groups that return matches only if the target string is followed or preceded by a particular character.

Lookaround assertions do not consume the characters in the input string or text. This makes them a "zero-width assertion", and that's why lookaround groups are also called "lookahead assertions".

There are two types of lookaround groups: **lookahead** and **lookbehind**. The two also have their positive and negative forms, so there are **positive lookahead**, **negative lookahead**, **positive lookbehind**, and **negative lookbehind** groups.

What is a Lookahead Group?

A lookahead group is a non-capturing group that lets you match a part of a string only if it is followed by another character in the string, without including that string or text to match in the pattern.

A lookahead group is useful when you want to match a string based on a condition. So, look at it like an `if` statement in a programming language.

There are two types of lookaheads, namely **positive lookahead** and **negative lookahead**.

Because you're still dealing with groupings, a positive lookahead is specified by an opening parenthesis followed by a question mark, an equal sign, the characters, and a closing parenthesis:

```
(?=chars)
```

For example, the pattern `x(=y)` means match `x` only if it is followed by `y`.

In the syntax of negative lookahead, you replace the equal sign with an exclamation mark:

```
(?!chars)
```

For example, the pattern `x(?!y)` means do not match `x` if it is followed by `y`.

Let's look at an example of a positive lookahead assertion.

Say you want to match the domain name of domains that have only the `.org` extension within a string of domains with other extensions. This pattern would do it:

```
[a-zA-Z]+(?:\.org)
```

In the pattern, `[a-zA-Z]+` represents one or more word characters, and `(?:\.org)` checks whether the domain contains a `.org` extension.

In the screenshot below, you can see that domain names that have a `.org` extension were matched:

org-matches

You also can see that the words "freeCodeCamp" and "catholic" were not included in the pattern, but they still matched the pattern because they have the `.org` extension.

If there are no domains with the `.org` extension in the target string, there won't be any match. That's true for the domains without the `.org` extension.

That way, you can extract text like that in JavaScript and do whatever you want with it:

```
const domains = 'koladechris.com freeCodeCamp.org mdn.com catholic.org';
const re = /[a-zA-Z]+(?:\.org)/g;

const charityWebsitesArr = domains.match(re);
const charityWebsites = charityWebsitesArr.join(',').replace(/,/g, ' and ');

console.log(charityWebsites, 'are examples of charity organizations.');//freeCodeCamp and
catholic are examples of charity organizations.
```

If you want to match the `.org` as well so the whole domain gets matched, you have to include the `.org` in the pattern:

org-all-match

Since lookahead groups don't consume characters, you will see a lot of developers use positive lookaheads to validate passwords.

Let's say you want the password to be at least six characters that includes a lowercase letter, an uppercase letter, a number, and a symbol. You can use lookaheads to define all of those conditions:

- `(?=.{6,})` → at least 6 characters
- `(?=.*[a-z])` - at least one lowercase character, but check if there are zero or many characters before it

- `(?=.*[A-Z])` – at least one lowercase character, but check if there are zero or many characters before it
- `(?=.*[0-9])` – at least one number, but check if there are zero or many characters before it
- `(?=.*[!@#$%^&*()+=-])` – accepted symbols, but check if there are zero or many characters before each
- `.*` – check if there are zero or many characters after the groups

Here's the full regular expression:

```
(?=.*{6,})(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*()+=- ]).*
```

And here is what matches the pattern and what does not:

pwdord-lookahead

To use that pattern in JavaScript, you can test it against a password string and do something from there:

```
const password = 'Tse23*';
const passwordRe =
  /^(?=.*{6,})(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*()+=- ]).*/;

if (passwordRe.test(password)) {
  console.log('Welcome to your dashboard!');
} else {
  throw new Error('Incorrect password!');
}

/**
output: Welcome to your dashboard!
*/
```

For the application of negative lookahead, it is useful when you don't want a certain character before the character(s) you are looking for in a string.

Let's say you want to extract all items of an array that do not have the article "the" before them. In that case, you can use the pattern below:

```
/^(?!.*\bThe\b).*$
```

In the pattern above:

- `^` ensures the regex pattern matches from the start of the line

- `(?!.*\bThe\b)` is the negative lookbehind that ensures that the article "the" is not in the target string
- `\bThe\b` is a word boundary that matches "The" and nothing else
- `.*` the wildcard that matches any character apart from a new line

```
let docTitles = [  
  'The Incredible Dr. Poll',  
  'Born in Africa',  
  "America's Funniest Home Videos",  
  'The Lion Queen',  
  'Snake in the City',  
];  
  
let re = /^(?!.*\bThe\b).*$;/;  
  
for (let title of docTitles) {  
  if (re.test(title)) {  
    console.log(`A Title without "The": ${title}`);  
  }  
}  
  
/*  
Output:  
A Title without "The": Born in Africa  
A Title without "The": America's funniest home videos  
A Title without "The": Snake in the City  
*/
```

What is a Lookbehind Group?

A lookbehind group is similar to lookahead group. But instead of checking if a certain character(s) follows what you're trying to match, it checks whether the character(s) precedes what you're trying to match.

So, a lookbehind group is a non-capturing group that lets you match a part of a string only if it is preceded by another character in the string, without including that string or text to match in the pattern.

Like lookaheads, there are also positive and negative lookbehind assertions. A positive lookbehind returns a match only if the character you want to match is preceded by another character you specify in your pattern. On the other hand, a negative lookbehind returns a match only if the character you want to match is not preceded by another character.

A positive lookahead is represented by an opening parenthesis, a question mark, a less than symbol, an equals sign, the character(s), and a closing parenthesis:

```
(?<=chars)
```

For example, the pattern `(?<=x)y` indicates you want to match `y` only if there's `x` before it. In this case, `xx` or `yx` won't match, but `xy` would match.

positive-lookbehind-match

For a negative lookahead, an exclamation mark replaces the equals sign:

```
(?<!=chars)
```

For example, the pattern `(?<!=x)y` means do not match `y` if there's `x` before it. In this case `by` would match, `my`, would match, but never `xy`.

negative-lkb-match

Positive lookahead groups can be useful for matching numbers preceded only by a certain currency symbol, for example numbers preceded by the dollar sign.

The regex pattern below has a positive lookahead that matches a number only if it is preceded by a dollar sign:

```
(?<=\\$)\\d+(\\.\\d*)?
```

In the pattern above, the lookahead `(?<=\\$)` checks whether there's a dollar sign before one or more digits (represented by `\\d+`). The other group, `(\\.\\d*)`, and the zero or one quantifier `(?)` check whether the number contains floating points.

Here's what matches and what does not:

positive-lkb-match

In JavaScript, what you can do with the numbers that match is to calculate the total with the `reduce()` method:

```
const myStr =  
  '10 pieces of the items cost $102.99, but you can get 15 for a discount of $2, and 20 for a  
  discount of $3.99';  
  
const re = /(?!<=\\$)\\d+(\\.\\d*)?/g;  
  
// put all the prices in an array
```

```
const allPrices = myStr.match(re); // [ '102.99', '2', '3.99' ]

// convert each of the prices to a number with map() and unary plus
const allPricesToNum = allPrices.map((price) => +price); // [ 102.99, 2, 3.99 ]

// add all the numbers with reduce()
const sumOfAllPrices = allPricesToNum.reduce((acc, curr) => acc + curr, 0); //
108.97999999999999

// add a dollar sign to the number and use toFixed() to round it down
console.log(`$${sumOfAllPrices.toFixed(2)}`); // $108.98
```

For the example of negative lookbehind, let's say you want to match a digit as long as it is not preceded by the dollar sign. This pattern does it:

```
(?<!\$)\d+
```

But unfortunately, it still looks out for a number inside another number and matches it even if there's a dollar sign before the whole number:

dollar-negative-lkb-err

To correct that behavior, you can surround the whole pattern with a word boundary (`\b`):

dollar-negative-lkb-fix

Negative lookbehind groups are supported in JavaScript as well:

```
const monies = '$123 456 $789 £12 ₺568 $8903 £345';
const re = /\b(?<!\$)\d+\b/g;

console.log('Monies without dollar sign:', monies.match(re)); // Monies without dollar sign: [
'456', '12', '568', '345' ]
```

Chapter 8: Regex Best Practices and Troubleshooting

Best Practices to Consider While Writing Regular Expressions

Over time, regular expressions can become complex and hard to understand, depending on the use case and purpose. Things may become more complicated if it takes you a long time to come back to them or you work in a team.

Luckily, there are a few best practices to consider while writing regular expressions so you can make things easier for yourself and your team members.

Here are those best practices:

- **Keep it simple and readable:** a simple, easy-to-read, and effective regex is better than a complex and effective regex. If you can make the regex efficient without using the complex concept of non-capturing groups like lookarounds (lookaheads and lookbehinds), then don't use them.
- **Avoid greedy matches:** metacharacters like `*` and `+` and the wildcard (`.`) are greedy by default. It's hard to do without them, but when you use them and they cause greediness, make sure you use the zero or one quantifier (`?`) on them. In addition, avoid using the wildcard where necessary.
- **Use comments to describe what a regex does:** if you're working in a team, try to explain what the regexes you write do so others can understand them without wasting time.
- **Use online regex testers:** instead of writing your regular expressions in your code editor, write them in regex testers where you can test what they match without writing some more code. Free online regex testers like [regex101](#) and [regexpal.com](#) also play a role in debugging because they can highlight errors and tell you what's wrong.
- **Escape special characters:** if you want to perform a literal match on metacharacters like `.`, `*`, `+`, `{`, `}`, and others, don't forget to escape them unless you're using them inside a character set. Sometimes, you even have to escape hyphens in a character set.

How to Write Accurate Regular Expressions

Writing accurate regular expressions with precision requires understanding what you want to match, the pattern to use, attention to detail, and an understanding of the underlying syntax and behavior of regular expressions in general.

This is crucial in order to ensure there are no avoidable errors and make sure the regexes you write effectively match the desired string.

Here are some tips to help you write accurate regular expressions:

- **Understand the string you want to match:** before you write the regex pattern to match a string, examine the string closely. Determine if you're targeting the whole string or a particular part of the string. If you're targeting a part of the string or you want to strip out some, look out for the pattern that you want follow. If you get familiar with the string, you can write a more accurate regex.
- **Be specific:** avoid using the wildcard where necessary. For instance, do not use the wildcard to match a number since you can use `\d` or `[0-9]`, or uppercase letters since

you can use `[A-Z]`.

- **Use quantifiers to shorten patterns:** if you want a particular part of your regex to match repeated occurrences, try to use quantifiers like `+`, `*`, `{n,m}`, `{n,}`, and `{n}`. For instance, if you want to match a date with `/` as the separator, you can use the pattern `\d{1,2}\d{1,2}\d{4}` instead of `\d\d\d\d\d\d\d\d`.
- **Use online regex testers:** online regex testers like regex101.com and regexpal.com help you write more accurate regexes by giving you a live match preview, highlighting matches, and showing you errors their engines encounter while processing the regexes.
- **Use word boundary to prevent unwanted matches:** surrounding your pattern with the word boundary (`\b`) can help you prevent unnecessary and unwanted matches. For example, if you want to match a 6-digit zip code, `\d{6}` can do it for you but will also match any part of the string that has 6 digits that follow one another. What would do it better is `\b\d{6}\b`.

Anchors (`^` and `$`) can also help prevent unwanted matches since they "anchor" a pattern to the start or end of the line. You can use them to make sure the match is found at the end or start of the line, or both.

For example:

- `/^Hello/i` would only match `Hello` or `hello` at the start of a line
- `/Hello$/i` would only match `Hello` or `hello` at the end of a line
- `/^Hello$/i` would only match `Hello` or `hello` if it's the only target string unless you have the multiline flag turned on and there's `Hello` or `hello` on another line.

If you have issues getting things right with a regex pattern, online testing tools like regex101.com and regexpal.com can also help you step through the pattern bit by bit. There are also regex visualizers you can use to check what's wrong with your regex patterns.

One of those tools that I find amazing is [Regulex \(jex.im/regulex\)](https://jex.im/regulex). It helps you put your regular expressions in a visual perspective you can export

regulex-right

And it can show you what goes wrong with your pattern:

regulex-wrong

Chapter 9: Applications of Regular Expressions

A Better Way to Match Dates

You've seen several patterns you can use to match dates in the `dd/mm/yyyy` format such as `\d\d\d\d\d\d\d\d\d\d\d\d`, `\d\d[/.-]\d\d[/.-]\d\d\d\d\d\d`, and `\d{1,2}\d{1,2}\d{4}`.

The problem is that those three patterns just check for the occurrence of a number, and not a valid date. For example, invalid dates `99/89/2022` or `42/32/1909` would still match those patterns:

invalid-date-matches

The solution is that you must account for the acceptable day, month, and year:

- the day can be 1 or 2 digits
- the day cannot exceed 31
- the month cannot exceed 12
- the year could be 2 or 4 digits, but never 1, 3, or greater than 4 digits

You should also account for:

- a day that could start with 0, 1, 2, or 3, but never 4 or greater
- a month that could start with 0, or 1, but never 2 or greater

Here's the regex pattern that satisfies those conditions:

```
/^(3[01]|[12][0-9]|0?[1-9])[-./](1[0-2]|0?[1-9])[-./](20[0-9]{2}|[0-9]{4}|[0-9]{2})$/gm
```

The image below is an illustration that labels each part of the pattern and explains what they do:

regexdate--1-

Here are the dates that match the pattern and those that do not:

date-re-matches

You can take the pattern and test it against some dates in JavaScript:

```
const re =  
  /^ (3[01]|[12][0-9]|0?[1-9]) [-./] (1[0-2]|0?[1-9]) [-./] (20[0-9]{2}|[0-9]{4}|[0-9]{2}) $/;  
  
function testDate(date) {  
  const dateTester = re.test(date);  
  console.log(dateTester);  
}  
  
testDate('12-01-2022'); // true  
testDate('31.11.1999'); // true  
testDate('02-01-21'); // true
```

```
testDate('42-01-2021'); // false
testDate('22-91-23'); // false
```

You can see the date, month, year, and separator parts of the pattern are in their respective groups. If you want to match other formats like `mm/dd/yyyy` or `yyyy/mm/dd`, you can twist the pattern around.

You can even make the pattern a little shorter by putting the first separator in a group and referencing it for the second separator:

```
^(3[01]|[12][0-9]|0?[1-9])([-./])(1[0-2]|0?[1-9])\2(20[0-9]{2}|[0-9]{4}|[0-9]{2})$
```

How to Match US Zip Codes

The zip codes in the US are a 5-digit number, but they may also have a 4-digit extension, for example, `56893` or `56893-9232`.

The pattern `\b\d{5}\b` would match a 5-digit zip-code:

zip-code-first-part-match

You also need to account for the other 4 digits and the hyphen between the two sets of numbers. The pattern, `\b\d{5}(\-\d{4})?\b` would do that for you.

Here's an image that labels each part of the pattern and explains what they do:

zip-regex--1-

You can also take the regex and extract all the zip codes that are matches:

```
const re = /\b\d{5}(\-\d{4})?\b/g;
const zipCodes = [
  '56893',
  'ca58392bn',
  '29043',
  '90342-9014',
  '89435',
  '75034',
  '90453-3056',
  '12345-6789',
  'b458923',
  '589323',
];
```

```

const matchedZipCodes = [];

for (const zipCode of zipCodes) {
  const matches = zipCode.match(re);
  if (matches) {
    matchedZipCodes.push(matches[0]);
  }
}

console.log(matchedZipCodes);

/*
Output:
[
  '56893',
  '29043',
  '90342-9014',
  '89435',
  '75034',
  '90453-3056',
  '12345-6789'
]
*/

```

And if you want the zip codes that are invalid, you can use the `filter()` array method to remove those that do not match the pattern:

```

const re = /\b\d{5}(\-\d{4})?\b/g;
const zipCodes = [
  '56893',
  'ca58392bn',
  '29043',
  '90342-9014',
  '89435',
  '75034',
  '90453-3056',
  '12345-6789',
  'b458923',
  '589323',

```

```
];

const invalidZipCodes = zipCodes.filter((zipCode) => !zipCode.match(re));

console.log(invalidZipCodes); // [ 'ca58392bn', 'b458923', '589323' ]
```

How to Match Email Addresses

Email addresses could be as simple as `john@email.com`, and as complex as you can ever imagine. So, there's no "one pattern" for validating email addresses. This also makes email validation a complex thing to do.

Validating emails with regex can also be a bit questionable because you can't stop anyone from making an email up. But still, there's a format you generally want the email address to be in whether it is made up or not. This is why you may want to use regular expressions to validate an email.

A pattern like `^\w{4,}@\w{3,}\.\w{3,}$` could be enough for validating simple and straightforward email addresses like `john@example.com`.

Here's an image that labels each part of the pattern and explains what they do:

simpleEmailRegExp

And here are the emails that match:

email-unreliable

As you can see, the pattern did not even match all the emails provided. That's because the pattern does not account for:

- emails with a period within usernames like `jane.doe@email.com`
- second-level domain (SLD) extensions like `john@example.abc.com`
- and country code second-level domains (ccSLDs) like `jane@email.co.uk`

In fact, a single email can even combine all the criteria listed above.

A better pattern for matching emails is `/^[\\w.-]+@[a-zA-Z\\d.-]+\\. [a-zA-Z]{2,}$`.

I also prepare an illustration that labels each part of the pattern and shows what they do:

betterEmailRegExp

This pattern matches an email address better than the first one:

email-a-bit-reliable

According to the RFC 5322 specification, the pattern that works 99% of the time for validating email is this:

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

N.B.: You should surround the pattern with anchors so it doesn't leave out a part of a possible email and match the others.

This is what I'm trying to point out:

matchingInvalidEmail

You can take that pattern into JavaScript and test it against some email addresses:

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

function matchEmail(email) {
  if (emailRe.test(email)) {
    console.log('Valid email!');
  } else {
    console.log('Invalid email');
  }
}

matchEmail('janedoe@email.com');
matchEmail('john.doe@email.com');
matchEmail('7@koala@email.com!');
matchEmail('kayla.simpson@email.co.uk');
matchEmail('kayla.simpson@email.co..uk');
```

As I pointed out earlier, matching email addresses with regex is a complex task. If you know the kind of email you'll be working with, it is better to tailor your regex for them.

Sometimes, to match an email, all you all you might need is a simple regex. Some other times, the pattern you need might be as complex as the one above.

How to Match Passwords

To match passwords, you can use a lookahead – since lookahead groups generally don't consume characters. But there are always multiple ways of doing the same thing in regular expressions, and programming in general of course.

You've seen a lookahead for matching 6-digit passwords already. This time around, let's say the password should not be less than 8 characters with at least one uppercase, one lowercase, one digit, and one symbol.

Here's the regex pattern that does just that:

```
^(?=.{8,})(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*()+=- ]).*
```

Here are the passwords it matches:

password-matches

You can take that into JavaScript and test it against possible passwords:

```
const passwordRe =
  /^(?=.{8,})(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*()+=- ]).*$/gm;

function matchPassWord(password) {
  if (passwordRe.test(password)) {
    console.log(true);
  } else {
    console.log(false);
  }
}

matchPassWord('johnDoe21^');
matchPassWord('Strong@123');
matchPassWord('weakpassword');
matchPassWord('ABcd12$');
matchPassWord('Longpassword1234!');
matchPassWord('Short@1');
matchPassWord('janEdoe34$');
```


You can also extract each of those group into its variable and test a password against it. This would let you show an error for that particular condition the password is trying to match:

```
const passwordLength = /^(?=.{8,})/,
    lowercaseChar = /^(?=.*[a-z])/,
    uppercaseChar = /^(?=.*[A-Z])/,
    numberChar = /^(?=.*[0-9])/,
    specialChar = /^(?=.*[!@#$%^&*()+=-~])/;

function validatePassword(password) {
  if (
    passwordLength.test(password) &&
    lowercaseChar.test(password) &&
    uppercaseChar.test(password) &&
    numberChar.test(password) &&
    specialChar.test(password)
  ) {
    console.log('Valid password!');
  } else {
    console.log('Invalid Password');
  }
}

validatePassword('johnDoe21^');
validatePassword('Strong@123');
validatePassword('weakpassword');
validatePassword('ABcd12$');
validatePassword('Longpassword1234!');
validatePassword('Short@1');
validatePassword('janEdoe34$');
```

Form Validation with Regex

One of the most popular ways developers use regular expressions is form validation. Since a form usually has input fields like name, email, password, and others, you can write a regular expression for what you expect the user to put in those input fields.

I prepared a little website where I show you how to validate the name, username, email, and password fields of a form with regex.

Here's the HTML:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="styles.css">
  <script src="form-validate.js" defer></script>
  <title>Form Validation with RegEx</title>
</head>

<body>

  <div id="error-message"></div>

  <form action="">

    <h1>Sign Up</h1>
    <p>Fill in the form fields</p>

    <div class="form-control">
      <label for="name">Name</label>
      <input type="text" name="name" id="name">
    </div>
    <div class="form-control">
      <label for="username">Username</label>
      <input type="text" name="username" id="username">
    </div>
    <div class="form-control">
      <label for="email">Email</label>
      <input type="email" name="email" id="email">
    </div>
    <div class="form-control">
      <label for="email">Password</label>
      <input type="password" name="password" id="password">
    </div>
    <input type="submit" value="Submit" id="submit">
  </form>

</body>
```

```
</html>
```

The CSS:

```
@import url('https://fonts.googleapis.com/css2?family=Roboto&display=swap');

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  background-color: #d0d0d5;
  color: #fff;
  font-family: 'Roboto', sans-serif;
}

form {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
  background-color: #3b3b4f;
  padding: 0.4rem 3rem 1rem;
  border-radius: 2px;
}

p {
  margin: 0.5rem 0;
}

#error-message {
  background-color: crimson;
  color: #fff;
  max-width: 80%;
  margin: 0.5rem auto 0;
  padding: 0.2rem 0.5rem;
  border-radius: 4px;
}
```

```
}

#error-message p {
  font-size: 14px;
  text-align: center;
}

.form-control {
  display: flex;
  flex-direction: column;
}

.form-control label {
  margin-bottom: 0.2rem;
}

.form-control input {
  width: 14rem;
  margin-bottom: 1.2rem;
  padding: 0.2rem;
  border: 2px solid #d0d0d5;
  border-radius: 2px;
}

.form-control input:focus {
  outline: none;
}

input[type='submit'] {
  background-color: #fecc4c;
  border-color: #f1a02a;
  font-family: 'Roboto', sans-serif;
  padding: 0.3rem;
  border-width: 1px;
  cursor: pointer;
}

input[type='submit']:hover {
  background-color: #e3bd53;
}
```

```

.hide {
  display: none;
}

.show {
  display: block;
}

@media screen and (max-width: 768px) {
  #error-message {
    margin: 0.5rem auto 0;
    padding: 0.1rem 0.2rem;
  }
}

@media screen and (max-width: 667px) {
  form {
    top: 61%;
  }

  #error-message {
    margin: 0.2rem auto 0;
    padding: 0.1rem 0.4rem;
  }
}

```

Most importantly, some well-commented JavaScript that contains the patterns I used, and how I tested each of the patterns against the respective fields they correlate with:

```

// Get the form element
const form = document.querySelector('form');
// Get the div element that shows the error(s)
const errorMessageDiv = document.querySelector('#error-message');

// The RegEx patterns in a "patterns" object
const patterns = {
  nameRe: /^[a-zA-Z]{2,35}\s[a-zA-Z]{2,35}$/ , // validates the name field
  usernameRe: /^[a-zA-Z]{3,30}(\d{1,4})?$/ , // validates the username field
  emailRe: /^[\\w.-]+@[a-zA-Z\\d.-]+\\. [a-zA-Z]{2,}$/ , // validates the email field

```

```

passwordRe:
    /^(?=.*{8,})(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*()+=-]).*$/, //
validates the password field
};

// Hide error message div when the page loads
errorMessageDiv.style.display = 'none';

// Add a submit event to the form
form.addEventListener('submit', validateAndSubmitForm);

// Form validation and submit function
function validateAndSubmitForm(e) {
    e.preventDefault();

    // Clear previous error messages
    errorMessageDiv.innerHTML = '';

    let nameInputValue = document.querySelector('#name').value;
    let usernameInputValue = document.querySelector('#username').value;
    let emailInputValue = document.querySelector('#email').value;
    let passwordInputValue = document.querySelector('#password').value;

    // Validate Name
    if (!patterns.nameRe.test(nameInputValue)) {
        showError('Name must have first name and last name separated by a space');
    }

    // Validate Username
    if (!patterns.usernameRe.test(usernameInputValue)) {
        showError(
            'Username must have between 3 and 30 characters and can include up to 4 digits at the
end'
        );
    }

    // Validate Email
    if (!patterns.emailRe.test(emailInputValue)) {
        showError('Enter a valid email address');
    }

```

```

// Validate Password
if (!patterns.passwordRe.test(passwordInputValue)) {
  showError(
    'Password must contain at least 8 characters, one lowercase letter, one uppercase
letter, one digit, and one special character.'
  );
}

// If there are no error messages, the form is valid, so you can submit it
if (errorMessageDiv.innerHTML === '') {
  console.log(nameInputValue);
  console.log(usernameInputValue);
  console.log(emailInputValue);
  console.log(passwordInputValue);

  // Hide the errorMessageDiv element since there are no errors
  errorMessageDiv.style.display = 'none';

  // Greet user
  alert(`Hi ${usernameInputValue} ☺☺☺\nThanks for filling this form`);

  // Clear input fields with the reset() method
  document.forms[0].reset();
} else {
  // Show the errorMessageDiv element if there are errors
  errorMessageDiv.style.display = 'block';
}
}

// The function responsible for showing error(s)
function showError(message) {
  const errorMessageElement = document.createElement('p');

  errorMessageElement.innerText = message;
  errorMessageDiv.appendChild(errorMessageElement);
}

```

This is what the form does:
form-validation

You can grab all the code in this [GitHub repo](#).

Article Table of Contents Generator

You can leverage the power of regular expressions to create a markdown table of contents generator.

Markdown tables of contents are made up of `h2` headings at the top level. Those `h2` headings have an `id` attribute you can use as the link. If you take a look at those `id` attributes, they are in the format below:

```
[How to Do ABC on XYZ!!!](##howtodoabconxyz)
```

This means you need to:

- use the text as it is as the link text and surround them with curly braces
- replace all spaces with an empty string
- replace all symbols with an empty string
- convert all the letters to lowercase
- surround the new link with parenthesis

The `replace()` and `toLowerCase()` string methods will help you achieve those things.

Here's the HTML for the app:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="styles.css">
  <script src="toc.js" defer></script>
  <title>TOC Generator</title>
</head>

<body>
  <div class="alert" id="alert">
    Please enter some heading texts!
  </div>

  <h1>Markdown Table of Content Generator for your Next Article</h1>
```



```

<h2>Paste in your headings to generate table of content</h2>
<form action="">

  <div class="form">
    <div class="form-control">
      <textarea name="toc" id="toc" cols="40" rows="15"></textarea>
    </div>

    <div class="form-control">
      <input type="submit" value="Generate" id="submit">
    </div>
  </div>
</form>

<div id="generated-toc">
  <!-- <p>Lorem ipsum dolor sit amet consectetur.</p>
  <p>Lorem ipsum dolor sit amet consectetur.</p>
  <p>Lorem ipsum dolor sit amet consectetur.</p>
  <p>Lorem ipsum dolor sit amet consectetur.</p>
  <p>Lorem ipsum dolor sit amet consectetur.</p>
  <p>Lorem ipsum dolor sit amet consectetur.</p>
  <p>Lorem ipsum dolor sit amet consectetur.</p> -->
</div>

</body>

</html>

```

The CSS:

```

@import url('https://fonts.googleapis.com/css2?family=Poppins&family=Roboto&display=swap');

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  font-family: 'Poppins' sans-serif;

```

```
background-color: #3b3b4f;
color: #fff;
}

h1 {
  margin-top: 2rem;
}

h1,
h2 {
  text-align: center;
  color: black;
  margin-bottom: 1rem;
  color: white;
}

form {
  max-width: 90%;
  margin: 0 auto;
  background-color: #d0d0d5;
  padding: 2rem;
  border-radius: 2px;
}

.form-control {
  text-align: center;
}

textarea {
  padding: 0.2rem 2rem 1rem 0.2rem;
}

textarea:focus {
  outline: 1px solid #3b3b4f;
}

input[type='submit'] {
  font-family: 'Poppins', sans-serif;
  font-size: 1.1rem;
  border: none;
```

```
background-color: #03732e;
color: #fff;
padding: 0.5rem 1rem;
border-radius: 4px;
margin-top: 1rem;
transition: 0.3s;
}
```

```
input[type='submit']:hover {
  cursor: pointer;
  background-color: #00471b;
}
```

```
#generated-toc {
  max-width: 60%;
  margin: 1rem auto;
  background-color: #d0d0d5;
  color: black;
  padding: 2rem;
  border-radius: 2px;
  text-align: left;
  font-size: 1.1rem;
  display: none;
}
```

```
.alert {
  display: none;
  margin: 1rem auto;
  max-width: 20%;
  text-align: center;
  padding: 1rem 0;
  border-radius: 2px;
  background-color: #eb7189;
  color: black;
}
```

```
@media screen and (max-width: 768px) {
  textarea {
    width: 16rem;
  }
}
```

```

.alert {
  max-width: 50%;
}
}

```

And the well-commented JavaScript:

```

const form = document.querySelector('form');
const generatedToc = document.querySelector('#generated-toc');
const alert = document.querySelector('.alert');

// Regular expressions to remove spaces and special characters
const spaceRe = /\s+/g;
const symRe = /[^\s*$%^&#@!.,©:&."=%' _\[\]-\|<>|÷™®)f({€¥¢–“”‘•~]/g;

function generateToc(e) {
  e.preventDefault();

  // Get the heading texts from the textarea
  const headingTexts = document.querySelector('#toc').value;

  if (headingTexts === '') {
    // Alert the user to enter heading texts
    alert.style.display = 'block';

    // hide the alert after 3 seconds
    setTimeout(() => {
      alert.style.display = 'none';
    }, 3000);

    // hide generated table of content (if any) since the user is trying to paste in another
    one
    generatedToc.style.display = 'none';
    return;
  }

  // Split the heading texts into an array of lines
  const headingLines = headingTexts.split('\n');

```

```

// Create an initial empty variable to save the table of content inside later
let tocContent = '';

// Loop through each line and generate the table of content items
headingLines.forEach((headingLine) => {
    // Remove any leading and/or trailing spaces from the line
    headingLine = headingLine.trim();

    // skip empty lines
    if (headingLine === '') {
        return;
    }

    // Generate the TOC link based on the heading text(s)
    const markdownLink = headingLine
        .replace(spaceRe, '') // replace spaces with an empty string
        .replace(symRe, '') // replace special characters (symbols)
        .toLowerCase(); // convert the link texts to lowercase characters

    // Create the table of contents item and append it to the tocContent variable
    tocContent += `<p>• [${headingLine}](${markdownLink})</p>`;
});

// Insert the generated table of contents into the "generated-toc" div element
generatedToc.innerHTML = tocContent;

// hide alert since there's currently no error at this point
alert.style.display = 'none';

// show the "generated-toc" div
generatedToc.style.display = 'block';

// clear the heading texts in the text area
document.querySelector('#toc').value = '';
}

// Add a submit event to the form
form.addEventListener('submit', generateToc);

/*

```

```
What is HTML?  
How to Contribute$ To Open Source Like a Boss!!  
Why you should Learn to C$ode in Java?  
  
Why you should get into Web3!  
Don't Attach Question Mark(?) to Hows!  
Stop Scaring Newbies!  
Why are you too cold&  
*/
```

Here's what's happening in the app:

tocgen

You can look through the code to have more understanding of how I was able to do that. The code is available on this [GitHub repo](#) and the app is [live here](#).

Glossary and References

Glossary of Terms

- **Regular Expression** or **RegEx**: A you can use for matching, searching, and manipulating text.
- **Pattern** or **regex pattern**: A sequence of characters that defines a search criterion in a regular expression.
- **Literal Character**: A character that matches itself in a regular expression (for example, "a" matches the character "a").
- **Flag**: Modifiers added after the closing delimiter of a regex to change matching behavior, such as **i** (case-insensitive) or **g** (global).
- **Metacharacter**: A character with a special meaning in a regular expression. Examples include **.** (any character), ***** (zero or more), and **|** (alternation).
- **Quantifier**: A metacharacter that specifies the number of repetitions of the preceding element. For example, ***** matches zero or more occurrences, and **{n}** matches **n** character(s).
- **Anchors**: Metacharacters that represent positions in the input string, such as **^** (start of line) and **\$** (end of line).
- **Grouping**: Using parentheses **()** to create a subexpression you can repeat or reference as a single unit.
- **Capture Group**: A group in a regular expression that captures and stores the matched text for later use.
- **Non-Capturing Group**: A group in a regular expression that matches the pattern but does not capture the matched text.

- **Greedy**: A matching behavior where quantifiers try to match as much as possible.
- **Lazy**: Another matching behavior where quantifiers match as little as possible. It is the opposite of **greedy**.
- **Lookahead**: A zero-width assertion that looks ahead to see if a pattern exists without including it in the match.
- **Lookbehind**: A zero-width assertion that looks behind to see if a pattern exists without including it in the match.
- **Escape Sequence and Character**: Using a backslash `\` to escape a metacharacter to treat it as a literal character. Or using it before a character to match its special meaning instead of the literal character. For example, `\d`.
- **Word Boundary**: A zero-width assertion that matches the position between a word character and a non-word character.
- **Negated Character Class**: A character class with `^` as the first character, matching any character not in the class.
- **Regex Engine**: The underlying software component that processes regular expressions and performs matching.
- **Case Sensitive**: A matching behavior where letters' cases must exactly match in the regex pattern and the input string.
- **Case Insensitive**: A flag (`i`) that enables case-insensitive matching in the regular expression.
- **Shorthand Character Class**: Shortcuts for common character classes, such as `\d` (digit), `\w` (word character), and `\s` (whitespace).
- **Backreference**: Referring to a captured group's content in the regex pattern. For example, `\1`.
- **Alternation**: Using the `|` metacharacter to match either of two patterns.
- **JavaScript RegExp Object**: The built-in JavaScript object that represents a regular expression. It has methods like `test()` and `exec()` for working with regular expressions.
- **Regular Expression Literals**: Regular expressions defined using slashes `/.../`, e.g., `/regex-pattern/`.
- **RegExp Constructor**: The RegExp constructor for creating regular expressions dynamically.

Quick Reference of Metacharacters and Quantifiers

- `\d`: matches any digit (0-9).
- `\D`: matches any non-digit character.
- `\w`: matches any word character (alphanumeric characters and underscore).
- `\W`: matches any non-word character.
- `\s`: matches any whitespace character (space, tab, newline, carriage return).
- `\S`: matches any non-whitespace character.
- `\b`: matches a word boundary position.
- `\B`: matches a non-word boundary position.
- `^`: matches the start of the line.

- `$`: matches the end of the line.
- `.`: matches any character except newline.
- `*`: matches zero or more occurrences.
- `+`: matches one or more occurrences.
- `?`: matches zero or one occurrence.
- `{n}`: matches exactly `n` (number) occurrences.
- `{n,}`: matches `n` or more occurrences.
- `{n,m}`: matches at least `n` and at most `m` (another number) occurrences.
- `|`: matches either the left or right expression.
- `(...)`: capturing group.
- `(?:...)`: non-capturing group.
- `\`: escapes a metacharacter in order to match it literally, or escapes a metacharacter that is also a literal character. For example, `\d`.
- `[...]`: character class.
- `[^...]`: negated character class.
- `(?=...)`: positive lookahead.
- `(?!...)`: negative lookahead.
- `(?<=...)`: positive lookbehind.
- `(?<!=...)`: negative lookbehind.

Thank you for reading!

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. [Get started](#)

Révision #1

Créé 28 juillet 2023 10:10:24 par Mickaël G.

Mis à jour 28 juillet 2023 10:16:49 par Mickaël G.