9

REGULAR EXPRESSIONS

Programming tools and techniques survive and spread in a chaotic, evolutionary way. It's not always the best or most brilliant ones that win, but rather the ones that function well enough within the right niche or that happen to be integrated with another successful piece of technology.

In this chapter, I will discuss one such tool, *regular expressions*. Regular expressions are a way to describe patterns in string data. They form a small, separate language that is part of JavaScript and many other languages and systems.

Regular expressions are both terribly awkward and extremely useful. Their syntax is cryptic and the programming interface JavaScript provides for them is clumsy. But they are a powerful tool for inspecting and processing strings. Properly understanding regular expressions will make you a more effective programmer.

Creating a Regular Expression

A regular expression is a type of object. It can be either constructed with the `RegExp` constructor or written as a literal value by enclosing a pattern in forward slash (`/`) characters.

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

Both of those regular expression objects represent the same pattern: an *a* character followed by a *b* followed by a *c*.

When using the `RegExp` constructor, the pattern is written as a normal string, so the usual rules apply for backslashes.

The second notation, where the pattern appears between slash characters, treats backslashes somewhat differently. First, since a forward slash ends the pattern, we need to put a backslash before any forward slash that we want to be *part* of the pattern. In addition, backslashes that aren't part of special character codes (like `\n`) will be *preserved*, rather than ignored as they are in strings, and change the meaning of the pattern. Some characters, such as question marks and plus signs, have special meanings in regular expressions and must be preceded by a backslash if they are meant to represent the character itself.

```
let aPlus = /A\+/;
```

Testing for Matches

Regular expression objects have a number of methods. The simplest one is `test`. If you pass it a string, it will return a Boolean telling you whether the string contains a match of the pattern in the expression.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

A regular expression consisting of only nonspecial characters simply represents that sequence of characters. If *abc* occurs anywhere in the string we are testing against (not just at the start), `test` will return `true`.

Sets of Characters

Finding out whether a string contains *abc* could just as well be done with a call to `indexOf`. Regular expressions are useful because they allow us to describe more complicated patterns.

Say we want to match any number. In a regular expression, putting a set of characters between square brackets makes that part of the expression match any of the characters between the brackets.

Both of the following expressions match all strings that contain a digit:

```
console.log(/[0123456789]/.test("in 1992"));
// → true
console.log(/[0-9]/.test("in 1992"));
// → true
```

Within square brackets, a hyphen (-) between two characters can be used to indicate a range of characters, where the ordering is determined by the character's Unicode number. Characters 0 to 9 sit right next to each other in this ordering (codes 48 to 57), so `[0-9]` covers all of them and matches any digit.

A number of common character groups have their own built-in shortcuts. Digits are one of them: `\d` means the same thing as `[0-9]`.

Any digit character

An alphanumeric character ("word character")

Any whitespace character (space, tab, newline, and similar)

A character that is *not* a digit

A nonalphanumeric character

A nonwhitespace character

Any character except for newline

You could match a date and time format like 01-30-2003 15:20 with the following expression:

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("01-30-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

That regular expression looks completely awful, doesn't it? Half of it is backslashes, producing a background noise that makes it hard to spot the actual pattern expressed. We'll see a slightly improved version of this expression later.

These backslash codes can also be used inside square brackets. For example, `[\d.]` means any digit or a period character. The period itself, between square brackets, loses its special meaning. The same goes for other special characters, such as the plus sign (`+`).

To *invert* a set of characters—that is, to express that you want to match any character *except* the ones in the set—you can write a caret (`^`) character after the opening bracket.

```
let nonBinary = /[^01]/;
console.log(nonBinary.test("1100100010100110"));
// → false
console.log(nonBinary.test("0111010112101001"));
// → true
```

International Characters

Because of JavaScript's initial simplistic implementation and the fact that this simplistic approach was later set in stone as standard behavior, Java-Script's regular expressions are rather dumb about characters that do not appear in the English language. For example, as far as JavaScript's regular expressions are concerned, a "word character" is only one of the 26 characters in the Latin alphabet (uppercase or lowercase), decimal digits, and, for some reason, the underscore character. Things like *é* or *β*, which most definitely are word characters, will not match `\w` (and *will* match uppercase `\W`, the nonword category).

By a strange historical accident, `\s` (whitespace) does not have this problem and matches all characters that the Unicode standard considers whitespace, including things like the nonbreaking space and the Mongolian vowel separator.

It is possible to use `\p` in a regular expression to match all characters to which the Unicode standard assigns a given property. This allows us to match things

like letters in a more cosmopolitan way. However, again due to compatibility with the original language standards, those are recognized only when you put a `u` character (for Unicode) after the regular expression.

	Any letter
	Any numeric character
	Any punctuation character
	Any nonletter (uppercase P inverts)
	Any character from the given script (see Chapter 5)

Using `\w` for text processing that may need to handle non-English text (or even English text with borrowed words like *cliché*) is a liability, since it won't treat characters like *é* as letters. Though they tend to be a bit more verbose, `\p` property groups are more robust.

```
console.log(/\p{L}/u.test("α"));
// → true
console.log(/\p{L}/u.test("!"));
// → false
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
```

On the other hand, if you are matching numbers in order to do something with them, you often do want `\d` for digits, since converting arbitrary numeric characters into a JavaScript number is not something that a function like `Number` can do for you.

Repeating Parts of a Pattern

We now know how to match a single digit. What if we want to match a whole number—a sequence of one or more digits?

When you put a plus sign (`+`) after something in a regular expression, it indicates that the element may be repeated more than once. Thus, `/\d+/` matches one or more digit characters.

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

The star (`*`) has a similar meaning but also allows the pattern to match zero times. Something with a star after it never prevents a pattern from matching—it'll just match zero instances if it can't find any suitable text to match.

A question mark (`?`) makes a part of a pattern *optional*, meaning it may occur zero times or one time. In the following example, the *u* character is allowed to occur, but the pattern also matches when it is missing:

```
let neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

To indicate that a pattern should occur a precise number of times, use braces. Putting `{4}` after an element, for example, requires it to occur exactly four times. It is also possible to specify a range this way: `{2,4}` means the element must occur at least twice and at most four times.

Here is another version of the date and time pattern that allows both single- and double-digit days, months, and hours. It is also slightly easier to decipher.

```
let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("1-30-2003 8:45"));
// → true
```

You can also specify open-ended ranges when using braces by omitting the number after the comma. For example, `{5,}` means five or more times.

Grouping Subexpressions

To use an operator like `*` or `+` on more than one element at a time, you must use parentheses. A part of a regular expression that is enclosed in parentheses counts as a single element as far as the operators following it are concerned.

```
let cartoonCrying = /boo+(hoo+)+/i;
console.log(cartoonCrying.test("Boohoooohoohooo"));
// → true
```

The first and second `+` characters apply only to the second `o` in `boo` and `hoo`, respectively. The third `+` applies to the whole group `(hoo+)`, matching one or more sequences like that.

The `i` at the end of the expression in the example makes this regular expression case insensitive, allowing it to match the uppercase *B* in the input string, even though the pattern is itself all lowercase.

Matches and Groups

The `test` method is the absolute simplest way to match a regular expression. It tells you only whether it matched and nothing else. Regular expressions also have an `exec` (execute) method that will return `null` if no match was found and return an object with information about the match otherwise.

```
let match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

An object returned from `exec` has an `index` property that tells us *where* in the string the successful match begins. Other than that, the object looks like (and in fact is) an array of strings, whose first element is the string that was matched. In the previous example, this is the sequence of digits that we were looking for.

String values have a `match` method that behaves similarly.

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

When the regular expression contains subexpressions grouped with parentheses, the text that matched those groups will also show up in the array. The whole match is always the first element. The next element is the part matched by the first group (the one whose opening parenthesis comes first in the expression), then the second group, and so on.

```
let quotedText = /'([^']*)'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

When a group does not end up being matched at all (for example, when followed by a question mark), its position in the output array will hold `undefined`. When a group is matched multiple times (for example, when followed by a `+`), only the last match ends up in the array.

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

If you want to use parentheses purely for grouping, without having them show up in the array of matches, you can put `?:` after the opening parenthesis.

```
console.log(/(?:na)+/.exec("banana"));
// → ["nana"]
```

Groups can be useful for extracting parts of a string. If we don't just want to verify whether a string contains a date but also extract it and construct an object that represents it, we can wrap parentheses around the digit patterns and directly pick the date out of the result of `exec`.

But first we'll take a brief detour to discuss the built-in way to represent date and time values in JavaScript.

The Date Class

JavaScript has a standard `Date` class for representing dates, or rather, points in time. If you simply create a date object using `new`, you get the current date and time.

```
console.log(new Date());
// → Fri Feb 02 2024 18:03:06 GMT+0100 (CET)
```

You can also create an object for a specific time.

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

JavaScript uses a convention where month numbers start at zero (so December is 11), yet day numbers start at one. This is confusing and silly. Be careful.

The last four arguments (hours, minutes, seconds, and milliseconds) are optional and taken to be zero when not given.

Timestamps are stored as the number of milliseconds since the start of 1970, in the UTC time zone. This follows a convention set by "Unix time," which was invented around that time. You can use negative numbers for times before 1970. The `getTime` method on a date object returns this number. It is big, as you can imagine.

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

If you give the `Date` constructor a single argument, that argument is treated as such a millisecond count. You can get the current millisecond count by

creating a new `Date` object and calling `getTime` on it or by calling the `Date.now` function.

Date objects provide methods such as `getFullYear`, `getMonth`, `getDate`, `getHours`, `getMinutes`, and `getSeconds` to extract their components. Besides `getFullYear` there's also `getYear`, which gives you the year minus 1900 (such as `98` or `125`) and is mostly useless.

Putting parentheses around the parts of the expression that we are interested in, we can now create a date object from a string.

```
function getDate(string) {
  let [_, month, day, year] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(year, month - 1, day);
}
console.log(getDate("1-30-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

The underscore (_) binding is ignored and used only to skip the full match element in the array returned by `exec`.

Boundaries and Look-Ahead

Unfortunately, `getDate` will also happily extract a date from the string "`100-1-30000`". A match may happen anywhere in the string, so in this case, it'll just start at the second character and end at the second-to-last character.

If we want to enforce that the match must span the whole string, we can add the markers `^` and `$`. The caret matches the start of the input string, whereas the dollar sign matches the end. Thus `/^\d+$/` matches a string consisting entirely of one or more digits, `/^!/` matches any string that starts with an exclamation mark, and `/x^/` does not match any string (there cannot be an `x` before the start of the string).

There is also a `\b` marker that matches *word boundaries*, positions that have a word character on one side and a nonword character on the other. Unfortunately, these use the same simplistic concept of word characters as `\w` and are therefore not very reliable.

Note that these boundary markers don't match any actual characters. They just enforce that a given condition holds at the place where it appears in the pattern.

Look-ahead tests do something similar. They provide a pattern and will make the match fail if the input doesn't match that pattern, but don't actually move the match position forward. They are written between `(?=` and `)`.

```
console.log(/a(?=e)/.exec("braeburn"));
// → ["a"]
console.log(/a(?! )/.exec("a b"));
// → null
```

The `e` in the first example is necessary to match, but is not part of the matched string. The `(?! )` notation expresses a *negative* look-ahead. This matches only if the pattern in the parentheses *doesn't* match, causing the second example to match only `a` characters that don't have a space after them.

Choice Patterns

Say we want to know whether a piece of text contains not only a number but a number followed by one of the words *pig*, *cow*, or *chicken*, or any of their plural forms.

We could write three regular expressions and test them in turn, but there is a nicer way. The pipe character (`|`) denotes a choice between the pattern to its left and the pattern to its right. We can use it in expressions like this:

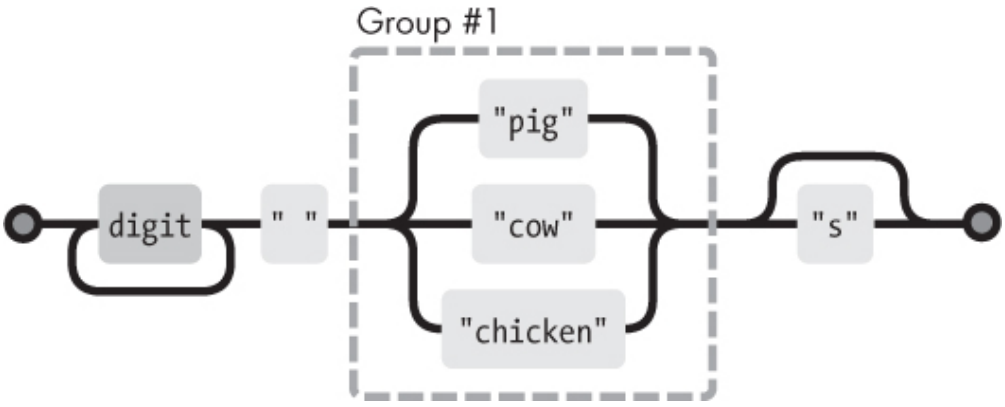
```
let animalCount = /\d+ (pig|cow|chicken)s?/;
console.log(animalCount.test("15 pigs"));
// → true
console.log(animalCount.test("15 pugs"));
// → false
```

Parentheses can be used to limit the part of the pattern to which the pipe operator applies, and you can put multiple such operators next to each other to express a choice between more than two alternatives.

The Mechanics of Matching

Conceptually, when you use `exec` or `test`, the regular expression engine looks for a match in your string by trying to match the expression first from the start of the string, then from the second character, and so on until it finds a match or reaches the end of the string. It'll either return the first match that can be found or fail to find any match at all.

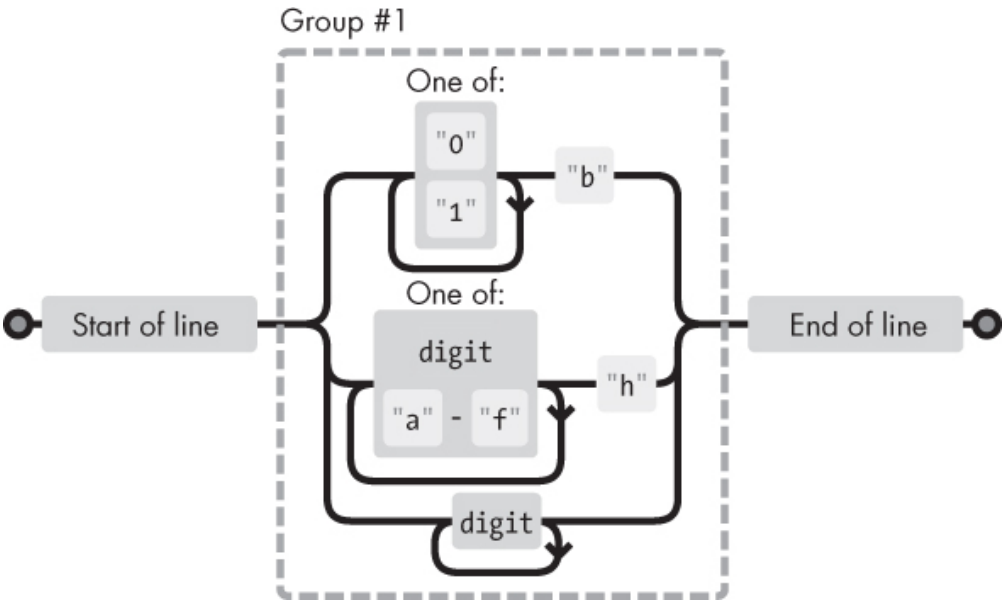
To do the actual matching, the engine treats a regular expression something like a flow diagram. This is the diagram for the livestock expression in the previous example:



If we can find a path from the left side of the diagram to the right side, our expression matches. We keep a current position in the string, and every time we move through a box, we verify that the part of the string after our current position matches that box.

Backtracking

The regular expression `/^([01]+b|[\da-f]+h|\d+)$/` matches either a binary number followed by a `b`, a hexadecimal number (that is, base 16, with the letters `a` to `f` standing for the digits 10 to 15) followed by an `h`, or a regular decimal number with no suffix character. This is the corresponding diagram:



When matching this expression, the top (binary) branch will often be entered even though the input does not actually contain a binary number. When matching the string "`103`", for example, it becomes clear only at the `3` that we are in the wrong branch. The string *does* match the expression, just not the branch we are currently in.

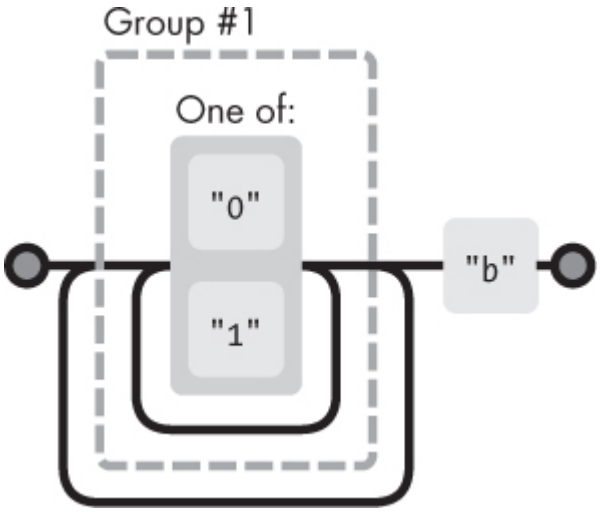
So the matcher *backtracks*. When entering a branch, it remembers its current position (in this case, at the start of the string, just past the first boundary box in the diagram) so that it can go back and try another branch if the current one does not work out. For the string "`103`", after encountering the `3` character, the matcher starts trying the branch for hexadecimal numbers, which fails again because there is no `h` after the number. It then tries the decimal number branch. This one fits, and a match is reported after all.

The matcher stops as soon as it finds a full match. This means that if multiple branches could potentially match a string, only the first one (ordered by where the branches appear in the regular expression) is used.

Backtracking also happens for repetition operators like + and `*`. If you match `/^.*x/` against "`abcxe`", the `.*` part will first try to consume the whole string. The engine will then realize that it needs an `x` to match the pattern. Since there is no `x` past the end of the string, the star operator tries to match one character less. But the matcher doesn't find an `x` after `abcx` either, so it backtracks again, matching the star operator to just `abc`. *Now* it finds an `x` where it needs it and reports a successful match from positions 0 to 4.

It is possible to write regular expressions that will do a *lot* of backtracking. This problem occurs when a pattern can match a piece of input in many

different ways. For example, if we get confused while writing a binary-number regular expression, we might accidentally write something like `/([01]+)+b/`.



If that tries to match some long series of zeros and ones with no trailing *b* character, the matcher first goes through the inner loop until it runs out of digits. Then it notices there is no *b*, so it backtracks one position, goes through the outer loop once, and gives up again, trying to backtrack out of the inner loop once more. It will continue to try every possible route through these two loops. This means the amount of work *doubles* with each additional character. For even just a few dozen characters, the resulting match will take practically forever.

The replace Method

String values have a `replace` method that can be used to replace part of the string with another string.

```
console.log("papa".replace("p", "m"));
// → mapa
```

The first argument can also be a regular expression, in which case the first match of the regular expression is replaced. When a `g` option (for *global*) is added after the regular expression, *all* matches in the string will be replaced, not just the first.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
```

```
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

The real power of using regular expressions with `replace` comes from the fact that we can refer to matched groups in the replacement string. For example, say we have a big string containing the names of people, one name per line, in the format *`Lastname, Firstname`*. If we want to swap these names and remove the comma to get a *`Firstname Lastname`* format, we can use the following code:

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nMilner, Robin"
    .replace(/(\p{L}+), (\p{L}+)/gu, "$2 $1"));
// → Barbara Liskov
//    John McCarthy
//    Robin Milner
```

The `$1` and `$2` in the replacement string refer to the parenthesized groups in the pattern. `$1` is replaced by the text that matched against the first group, `$2` by the second, and so on, up to `$9`. The whole match can be referred to with `$&`.

It is possible to pass a function—rather than a string—as the second argument to `replace`. For each replacement, the function will be called with the matched groups (as well as the whole match) as arguments, and its return value will be inserted into the new string.

Here's an example:

```
let stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) { // Only one left, remove the 's'
    unit = unit.slice(0, unit.length - 1);
  } else if (amount == 0) {
    amount = "no";
  }
  return amount + " " + unit;
}
console.log(stock.replace(/(\d+) (\p{L}+)/gu, minusOne)
// → no lemon, 1 cabbage, and 100 eggs
```

This code takes a string, finds all occurrences of a number followed by an alphanumeric word, and returns a string that has one less of every such quantity.

The `(\d+)` group ends up as the `amount` argument to the function, and the `(\p{L}+)` group gets bound to `unit`. The function converts `amount` to a number—which always works, since it matched `\d+` earlier—and makes some adjustments in case there is only one or zero left.

Greed

We can use `replace` to write a function that removes all comments from a piece of JavaScript code. Here is a first attempt:

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(stripComments("1 + /* 2 */3"));
// → 1 + 3
console.log(stripComments("x = 10;// ten!"));
// → x = 10;
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1  1
```

The part before the `|` operator matches two slash characters followed by any number of non-newline characters. The part for multiline comments is more involved. We use `[^]` (any character that is not in the empty set of characters) as a way to match any character. We cannot just use a period here because block comments can continue on a new line, and the period character does not match newline characters.

But the output for the last line appears to have gone wrong. Why?

The `[^]*` part of the expression, as I described in the section on backtracking, will first match as much as it can. If that causes the next part of the pattern to fail, the matcher moves back one character and tries again from there. In the example, the matcher first tries to match the whole rest of the string and then moves back from there. It will find an occurrence of `*/` after going back four characters and match that. This is not what we wanted—the intention was to

match a single comment, not to go all the way to the end of the code and find the end of the last block comment.

Because of this behavior, we say the repetition operators (`+`, `*`, `?`, and `{}`) are *greedy*, meaning they match as much as they can and backtrack from there. If you put a question mark after them (`+?`, `*?`, `??`, `{}?`), they become nongreedy and start by matching as little as possible, matching more only when the remaining pattern does not fit the smaller match.

And that is exactly what we want in this case. By having the star match the smallest stretch of characters that brings us to a `*/`, we consume one block comment and nothing more.

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

A lot of bugs in regular expression programs can be traced to unintentionally using a greedy operator where a nongreedy one would work better. When using a repetition operator, prefer the nongreedy variant.

Dynamically Creating RegExp Objects

In some cases, you may not know the exact pattern you need to match against when you are writing your code. Say you want to test for the user's name in a piece of text. You can build up a string and use the `RegExp` constructor on that.

```
let name = "harry";
let regexp = new RegExp("(^|\\s)" + name + "($|\\s)", "
console.log(regexp.test("Harry is a dodgy character."))
// → true
```

When creating the `\s` part of the string, we have to use two backslashes because we are writing them in a normal string, not a slash-enclosed regular expression. The second argument to the `RegExp` constructor contains the options for the regular expression—in this case, "`gi`" for global and case insensitive.

But what if the name is "`dea+hl[]rd`" because our user is a nerdy teenager? That would result in a nonsensical regular expression that won't actually match the user's name.

To work around this, we can add backslashes before any character that has a special meaning.

```
let name = "dea+hl[]rd";
let escaped = name.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("(^|\\s)" + escaped + "($|\\s)"
let text = "This dea+hl[]rd guy is super annoying.";
console.log(regexp.test(text));
// → true
```

The search Method

While the `indexOf` method on strings cannot be called with a regular expression, there is another method, `search`, that does expect a regular expression. Like `indexOf`, it returns the first index on which the expression was found, or `-1` when it wasn't found.

```
console.log("  word".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

Unfortunately, there is no way to indicate that the match should start at a given offset (like we can with the second argument to `indexOf`), which would often be useful.

The lastIndex Property

The `exec` method similarly does not provide a convenient way to start searching from a given position in the string. But it does provide an *in*convenient way.

Regular expression objects have properties. One such property is `source`, which contains the string that expression was created from. Another property

is `lastIndex`, which controls, in some limited circumstances, where the next match will start.

Those circumstances are that the regular expression must have the global (`g`) or sticky (`y`) option enabled, and the match must happen through the `exec` method. Again, a less confusing solution would have been to just allow an extra argument to be passed to `exec`, but confusion is an essential feature of JavaScript's regular expression interface.

```
let pattern = /y/g;
pattern.lastIndex = 3;
let match = pattern.exec("xyzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

If the match was successful, the call to `exec` automatically updates the `lastIndex` property to point after the match. If no match was found, `lastIndex` is set back to 0, which is also the value it has in a newly constructed regular expression object.

The difference between the global and the sticky options is that when sticky is enabled, the match will succeed only if it starts directly at `lastIndex`, whereas with global, it will search ahead for a position where a match can start.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let sticky = /abc/y;
console.log(sticky.exec("xyz abc"));
// → null
```

When using a shared regular expression value for multiple `exec` calls, these automatic updates to the `lastIndex` property can cause problems. Your regular expression might be accidentally starting at an index left over from a previous call.

```
let digit = /\d/g;
console.log(digit.exec("here it is: 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

Another interesting effect of the global option is that it changes the way the `match` method on strings works. When called with a global expression, instead of returning an array similar to that returned by `exec`, `match` will find *all* matches of the pattern in the string and return an array containing the matched strings.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

So be cautious with global regular expressions. The cases where they are necessary—calls to `replace` and places where you want to explicitly use `lastIndex`—are typically the only situations where you want to use them.

A common thing to do is to find all the matches of a regular expression in a string. We can do this by using the `matchAll` method.

```
let input = "A string with 3 numbers in it... 42 and 88
let matches = input.matchAll(/\d+/g);
for (let match of matches) {
  console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
//    Found 42 at 33
//    Found 88 at 40
```

This method returns an array of match arrays. The regular expression given to `matchAll` *must* have `g` enabled.

Parsing an INI File

To conclude the chapter, we'll look at a problem that calls for regular expressions. Imagine we are writing a program to automatically collect

information about our enemies from the internet. (We will not actually write that program here, just the part that reads the configuration file. Sorry.) The configuration file looks like this:

```
searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7
; Comments are preceded by a semicolon...
; Each section concerns an individual enemy
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
fullname=Davaeorn
type=evil wizard
outputdir=/home/marijn/enemies/davaeorn
```

The exact rules for this format—which is a widely used file format, usually called an *INI* file—are as follows:

- Blank lines and lines starting with semicolons are ignored.
- Lines wrapped in `[` and `]` start a new section.
- Lines containing an alphanumeric identifier followed by an `=` character add a setting to the current section.
- Anything else is invalid.

Our task is to convert a string like this into an object whose properties hold strings for settings written before the first section header and subobjects for sections, with those subobjects holding the section's settings.

Since the format has to be processed line by line, splitting up the file into separate lines is a good start. We saw the `split` method in Chapter 4. Some operating systems, however, use not just a newline character to separate lines but a carriage return character followed by a newline (`"\r\n"`). Given that the `split` method also allows a regular expression as its argument, we can use a regular expression like `/\r?\n/` to split in a way that allows both `"\n"` and `"\r\n"` between lines.

```
function parseINI(string) {
  // Start with an object to hold the top-level fields
  let result = {};
  let section = result;
  for (let line of string.split(/\r?\n/)) {
    let match;
    if (match = line.match(/^(\w+)=(.*)$/)) {
      section[match[1]] = match[2];
    } else if (match = line.match(/^\[(.*)\]$/)) {
      section = result[match[1]] = {};
    } else if (!/^\s*(;|$)/.test(line)) {
      throw new Error("Line '" + line + "' is not valic
    }
  };
  return result;
}

console.log(parseINI(`
name=Vasilis
[address]
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

The code goes over the file's lines and builds up an object. Properties at the top are stored directly into that object, whereas properties found in sections are stored in a separate section object. The `section` binding points at the object for the current section.

There are two kinds of significant lines—section headers or property lines. When a line is a regular property, it is stored in the current section. When it is a section header, a new section object is created, and `section` is set to point at it.

Note the recurring use of `^` and `$` to make sure the expression matches the whole line, not just part of it. Leaving these out results in code that mostly works but behaves strangely for some input, which can be a difficult bug to track down.

The pattern `if (match = string.match(...))` makes use of the fact that the value of an assignment expression (`=`) is the assigned value. You often aren't sure that your call to `match` will succeed, so you can access the resulting object only

inside an `if` statement that tests for this. To not break the pleasant chain of `else if` forms, we assign the result of the match to a binding and immediately use that assignment as the test for the `if` statement.

If a line is not a section header or a property, the function checks whether it is a comment or an empty line using the expression `/^\s*(;|$)/` to match lines that contain either only whitespace, or whitespace followed by a semicolon (making the rest of the line a comment). When a line doesn't match any of the expected forms, the function throws an exception.

Code Units and Characters

Another design mistake that's been standardized in JavaScript regular expressions is that by default, operators like `.` or `?` work on code units (as discussed in Chapter 5), not actual characters. This means characters that are composed of two code units behave strangely.

```
console.log(/
                          
{3}/.test("
                          
                          
                          
"));
// → false
console.log(/<.>/.test("<
                          
>"));
// → false
console.log(/<.>/u.test("<
                          
>"));
// → true
```

The problem is that the

in the first line is treated as two code units, and `{3}` is applied only to the second unit. Similarly, the dot matches a single code unit, not the two that make up the rose emoji.

You must add the `u` (Unicode) option to your regular expression to make it treat such characters properly.

```
console.log(/
                          🍎
{3}/u.test("
                          🍎
                          🍎
                          🍎
"));
// → true
```

Summary

Regular expressions are objects that represent patterns in strings. They use their own language to express these patterns.

	A sequence of characters
	Any character from a set of characters
	Any character *not* in a set of characters
	Any character in a range of characters
	One or more occurrences of the pattern `x`
	One or more occurrences, nongreedy
	Zero or more occurrences
	Zero or one occurrence

Two to four occurrences

A group

Any one of several patterns

Any digit character

An alphanumeric character (“word character”)

Any whitespace character

Any character except newlines

Any letter character

Start of input

End of input

A look-ahead test

A regular expression has a method `test` to test whether a given string matches it. It also has a method `exec` that, when a match is found, returns an array containing all matched groups. Such an array has an `index` property that indicates where the match started.

Strings have a `match` method to match them against a regular expression and a `search` method to search for one, returning only the starting position of the match. Their `replace` method can replace matches of a pattern with a replacement string or function.

Regular expressions can have options, which are written after the closing slash. The `i` option makes the match case insensitive. The `g` option makes the expression *global*, which, among other things, causes the `replace` method to replace all instances instead of just the first. The `y` option makes an expression sticky, which means that it will not search ahead and skip part of the string when looking for a match. The `u` option turns on Unicode mode, which

enables `\p` syntax and fixes a number of problems around the handling of characters that take up two code units.

Regular expressions are a sharp tool with an awkward handle. They simplify some tasks tremendously but can quickly become unmanageable when applied to complex problems. Part of knowing how to use them is resisting the urge to try to shoehorn things into them that they cannot cleanly express.

Exercises

It is almost unavoidable that, in the course of working on these exercises, you will get confused and frustrated by some regular expression's inexplicable behavior. Sometimes it helps to enter your expression into an online tool like *https://www.debuggex.com* to see whether its visualization corresponds to what you intended and to experiment with the way it responds to various input strings.

Regexp Golf

Code golf is a term used for the game of trying to express a particular program in as few characters as possible. Similarly, *regexp golf* is the practice of writing as tiny a regular expression as possible to match a given pattern and *only* that pattern.

For each of the following items, write a regular expression to test whether the given pattern occurs in a string. The regular expression should match only strings containing the pattern. When your expression works, see whether you can make it any smaller.

1. *car* and *cat*
2. *pop* and *prop*
3. *ferret*, *ferry*, and *ferrari*
4. Any word ending in *ious*
5. A whitespace character followed by a period, comma, colon, or semicolon
6. A word longer than six letters
7. A word without the letter *e* (or *E*)

Refer to the table in the chapter summary for help. Test each solution with a few test strings.

Quoting Style

Imagine you have written a story and used single quotation marks throughout to mark pieces of dialogue. Now you want to replace all the dialogue quotes with double quotes, while keeping the single quotes used in contractions like *aren't*.

Think of a pattern that distinguishes these two kinds of quote usage and craft a call to the `replace` method that does the proper replacement.

Numbers Again

Write an expression that matches only JavaScript-style numbers. It must support an optional minus *or* plus sign in front of the number, the decimal dot, and exponent notation—`5e-3` or `1E10`—again with an optional sign in front of the exponent. Also note that it is not necessary for there to be digits in front of or after the dot, but the number cannot be a dot alone. That is, `.5` and `5.` are valid JavaScript numbers, but a lone dot isn't.