**Chapter 9**

**Regular Expressions**

Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.’ Now they have two problems.

Jamie Zawinski

Yuan-Ma said, ‘When you cut against the grain of the wood, much strength is needed. When you program against the grain of a problem, much code is needed.’

Master Yuan-Ma, *The Book of Programming*

Programming tools and techniques survive and spread in a chaotic, evolutionary way. It’s not always the pretty or brilliant ones that win but rather the ones that function well enough within the right niche—for example, by being 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 tools.

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 either be constructed with the RegExp constructor or written as a literal value by enclosing the pattern in forward slash (/) characters.

var re1 = new RegExp("abc");

var re2 = /abc/;

Both of these 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.

var eighteenPlus = /eighteen\+/;

Knowing precisely what characters to backslash-escape when writing regular expressions requires you to know every character with a special meaning. For the time being, this may not be realistic, so when in doubt, just put a backslash before any character that is not a letter, number, or whitespace.

**Testing for matches**

Regular expression objects have a number of methods. The simplest one istest. 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.

**Matching a set of characters**

Finding out whether a string contains *abc* could just as well be done with a call to indexOf. Regular expressions allow us to go beyond that and express 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 dash (-) 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.

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

|  |  |
| --- | --- |
| \d | Any digit character |
| \w | An alphanumeric character (“word character”) |
| \s | Any whitespace character (space, tab, newline, and similar) |
| \D | A character that is *not* a digit |
| \W | A nonalphanumeric character |
| \S | A nonwhitespace character |
| . | Any character except for newline |

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

var dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;

console.log(dateTime.test("30-01-2003 15:20"));

// → true

console.log(dateTime.test("30-jan-2003 15:20"));

// → false

That looks completely awful, doesn’t it? It has way too many backslashes, producing background noise that makes it hard to spot the actual pattern expressed. We’ll see a slightly improved version of this expression [later](http://eloquentjavascript.net/09_regexp.html#date_regexp_counted).

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

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.

var notBinary = /[^01]/;

console.log(notBinary.test("1100100010100110"));

// → false

console.log(notBinary.test("1100100010200110"));

// → true

**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 or one time. In the following example, the *u* character is allowed to occur, but the pattern also matches when it is missing.

var 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 curly 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 more readable.

var dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;

console.log(dateTime.test("30-1-2003 8:45"));

// → true

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

**Grouping subexpressions**

To use an operator like \* or + on more than one element at a time, you can 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.

var 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 previous 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.

var 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.

var 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. Similarly, when a group is matched multiple times, only the last match ends up in the array.

console.log(/bad(ly)?/.exec("bad"));

// → ["bad", undefined]

console.log(/(\d)+/.exec("123"));

// → ["123", "3"]

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, a brief detour, in which we discuss the preferred way to store date and time values in JavaScript.

**The date type**

JavaScript has a standard object type for representing dates—or rather, points in time. It is called Date. If you simply create a date object using new, you get the current date and time.

console.log(new Date());

// → Wed Dec 04 2013 14:24:57 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, using negative numbers for times before 1970 (following a convention set by “Unix time”, which was invented around that time). 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 but also by calling the Date.now function.

Date objects provide methods like getFullYear, getMonth, getDate, getHours, getMinutes, and getSeconds to extract their components. There’s also getYear, which gives you a rather useless two-digit year value (such as 93 or 14).

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

function findDate(string) {

var dateTime = /(\d{1,2})-(\d{1,2})-(\d{4})/;

var match = dateTime.exec(string);

return new Date(Number(match[3]),

Number(match[2]) - 1,

Number(match[1]));

}

console.log(findDate("30-1-2003"));

// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)

**Word and string boundaries**

Unfortunately, findDate will also happily extract the nonsensical date 00-1-3000 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 $. The caret matches the start of the input string, while the dollar sign matches the end. So, /^\d+$/ 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).

If, on the other hand, we just want to make sure the date starts and ends on a word boundary, we can use the marker \b. A word boundary can be the start or end of the string or any point in the string that has a word character (as in\w) on one side and a nonword character on the other.

console.log(/cat/.test("concatenate"));

// → true

console.log(/\bcat\b/.test("concatenate"));

// → false

Note that a boundary marker doesn’t represent an actual character. It just enforces that the regular expression matches only when a certain condition holds at the place where it appears in the pattern.

**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. So I can say this:

var animalCount = /\b\d+ (pig|cow|chicken)s?\b/;

console.log(animalCount.test("15 pigs"));

// → true

console.log(animalCount.test("15 pigchickens"));

// → false

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

**The mechanics of matching**

Regular expressions can be thought of as flow diagrams. This is the diagram for the livestock expression in the previous example:

Our expression matches a string if we can find a path from the left side of the diagram to the right side. 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.

So if we try to match "the 3 pigs" with our regular expression, our progress through the flow chart would look like this:

* At position 4, there is a word boundary, so we can move past the first box.
* Still at position 4, we find a digit, so we can also move past the second box.
* At position 5, one path loops back to before the second (digit) box, while the other moves forward through the box that holds a single space character. There is a space here, not a digit, so we must take the second path.
* We are now at position 6 (the start of “pigs”) and at the three-way branch in the diagram. We don’t see “cow” or “chicken” here, but we do see “pig”, so we take that branch.
* At position 9, after the three-way branch, one path skips the *s* box and goes straight to the final word boundary, while the other path matches an *s*. There is an *s* character here, not a word boundary, so we go through the *s* box.
* We’re at position 10 (the end of the string) and can match only a word boundary. The end of a string counts as a word boundary, so we go through the last box and have successfully matched this string.

Conceptually, a regular expression engine looks for a match in a string as follows: it starts at the start of the string and tries a match there. In this case, there *is* a word boundary there, so it’d get past the first box—but there is no digit, so it’d fail at the second box. Then it moves on to the second character in the string and tries to begin a new match there... and so on, until it finds a match or reaches the end of the string and decides that there really is no match.

**Backtracking**

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

When matching this expression, it will often happen that the top (binary) branch is 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, it will start trying the branch for decimal numbers. This one matches, so 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 will first go 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, which 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 to 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

It would have been sensible if the choice between replacing one match or all matches was made through an additional argument to replace or by providing a different method, replaceAll. But for some unfortunate reason, the choice relies on a property of the regular expression instead.

The real power of using regular expressions with replace comes from the fact that we can refer back 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 simple Firstname Lastname format, we can use the following code:

console.log(

"Hopper, Grace\nMcCarthy, John\nRitchie, Dennis"

.replace(/([\w ]+), ([\w ]+)/g, "$2 $1"));

// → Grace Hopper

// John McCarthy

// Dennis Ritchie

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, $2by the second, and so on, up to $9. The whole match can be referred to with $&.

It is also 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 a simple example:

var s = "the cia and fbi";

console.log(s.replace(/\b(fbi|cia)\b/g, function(str) {

return str.toUpperCase();

}));

// → the CIA and FBI

And here’s a more interesting one:

var 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+) (\w+)/g, minusOne));

// → no lemon, 1 cabbage, and 100 eggs

This takes a string, finds all occurrences of a number followed by an alphanumeric word, and returns a string wherein every such occurrence is decremented by one.

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

**Greed**

It isn’t hard to 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 *or* operator simply 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 dot here because block comments can continue on a new line, and dots do not match the newline character.

But the output of the previous example 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, consider the nongreedy variant first.

**Dynamically creating RegExp objects**

There are cases where you might not know the exact pattern you need to match against when you are writing your code. Say you want to look for the user’s name in a piece of text and enclose it in underscore characters to make it stand out. Since you will know the name only once the program is actually running, you can’t use the slash-based notation.

But you can build up a string and use the RegExp constructor on that. Here’s an example:

var name = "harry";

var text = "Harry is a suspicious character.";

var regexp = new RegExp("\\b(" + name + ")\\b", "gi");

console.log(text.replace(regexp, "\_$1\_"));

// → \_Harry\_ is a suspicious character.

When creating the \b boundary markers, 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, which won’t actually match the user’s name.

To work around this, we can add backslashes before any character that we don’t trust. Adding backslashes before alphabetic characters is a bad idea because things like \b and \n have a special meaning. But escaping everything that’s not alphanumeric or whitespace is safe.

var name = "dea+hl[]rd";

var text = "This dea+hl[]rd guy is super annoying.";

var escaped = name.replace(/[^\w\s]/g, "\\$&");

var regexp = new RegExp("\\b(" + escaped + ")\\b", "gi");

console.log(text.replace(regexp, "\_$1\_"));

// → This \_dea+hl[]rd\_ guy is super annoying.

**The search method**

The indexOf method on strings cannot be called with a regular expression. But there is another method, search, which 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) option enabled, and the match must happen through the exec method. Again, a more sane solution would have been to just allow an extra argument to be passed to exec, but sanity is not a defining characteristic of JavaScript’s regular expression interface.

var pattern = /y/g;

pattern.lastIndex = 3;

var 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 zero, which is also the value it has in a newly constructed regular expression object.

When using a global 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 that was left over from a previous call.

var 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 places where you want to use them.

**Looping over matches**

A common pattern is to scan through all occurrences of a pattern in a string, in a way that gives us access to the match object in the loop body, by usinglastIndex and exec.

var input = "A string with 3 numbers in it... 42 and 88.";

var number = /\b(\d+)\b/g;

var match;

while (match = number.exec(input))

console.log("Found", match[1], "at", match.index);

// → Found 3 at 14

// Found 42 at 33

// Found 88 at 40

This makes use of the fact that the value of an assignment expression (=) is the assigned value. So by using match = number.exec(input) as the condition in the while statement, we perform the match at the start of each iteration, save its result in a variable, and stop looping when no more matches are found.

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

searchengine=http://www.google.com/search?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

[gargamel]

fullname=Gargamel

type=evil sorcerer

outputdir=/home/marijn/enemies/gargamel

The exact rules for this format (which is actually a widely used 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 array of objects, each with a name property and an array of settings. We’ll need one such object for each section and one for the global settings at the top.

Since the format has to be processed line by line, splitting up the file into separate lines is a good start. We used string.split("\n") to do this in [Chapter 6](http://eloquentjavascript.net/06_object.html#split). 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 split on 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

var currentSection = {name: null, fields: []};

var categories = [currentSection];

string.split(/\r?\n/).forEach(function(line) {

var match;

if (/^\s\*(;.\*)?$/.test(line)) {

return;

} else if (match = line.match(/^\[(.\*)\]$/)) {

currentSection = {name: match[1], fields: []};

categories.push(currentSection);

} else if (match = line.match(/^(\w+)=(.\*)$/)) {

currentSection.fields.push({name: match[1],

value: match[2]});

} else {

throw new Error("Line '" + line + "' is invalid.");

}

});

return categories;

}

This code goes over every line in the file, updating the “current section” object as it goes along. First, it checks whether the line can be ignored, using the expression /^\s\*(;.\*)?$/. Do you see how it works? The part between the parentheses will match comments, and the ? will make sure it also matches lines containing only whitespace.

If the line is not a comment, the code then checks whether the line starts a new section. If so, it creates a new current section object, to which subsequent settings will be added.

The last meaningful possibility is that the line is a normal setting, which the code adds to the current section object.

If a line matches none of these forms, the function throws an error.

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(...)) is similar to the trick of using an assignment as the condition for while. 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 if forms, we assign the result of the match to a variable and immediately use that assignment as the test in the if statement.

**International characters**

Because of JavaScript’s initial simplistic implementation and the fact that this simplistic approach was later set in stone as standard behavior, JavaScript’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) 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.

Some regular expression implementations in other programming languages have syntax to match specific Unicode character categories, such as “all uppercase letters”, “all punctuation”, or “control characters”. There are plans to add support for such categories JavaScript, but it unfortunately looks like they won’t be realized in the near future.

**Summary**

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

|  |  |
| --- | --- |
| /abc/ | A sequence of characters |
| /[abc]/ | Any character from a set of characters |
| /[^abc]/ | Any character *not* in a set of characters |
| /[0-9]/ | Any character in a range of characters |
| /x+/ | One or more occurrences of the pattern x |
| /x+?/ | One or more occurrences, nongreedy |
| /x\*/ | Zero or more occurrences |
| /x?/ | Zero or one occurrence |
| /x{2,4}/ | Between two and four occurrences |
| /(abc)/ | A group |
| /a|b|c/ | Any one of several patterns |
| /\d/ | Any digit character |
| /\w/ | An alphanumeric character (“word character”) |
| /\s/ | Any whitespace character |
| /./ | Any character except newlines |
| /\b/ | A word boundary |
| /^/ | Start of input |
| /$/ | End of input |

A regular expression has a method test to test whether a given string matches it. It also has an exec method 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. Alternatively, you can pass a function to replace, which will be used to build up a replacement string based on the match text and matched groups.

Regular expressions can have options, which are written after the closing slash. The i option makes the match case insensitive, while the g option makes the expression *global*, which, among other things, causes the replacemethod to replace all instances instead of just the first.

The RegExp constructor can be used to create a regular expression value from a string.

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 that they cannot sanely express into them.

**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[*debuggex.com*](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 any of the given substrings occur in a string. The regular expression should match only strings containing one of the substrings described. Do not worry about word boundaries unless explicitly mentioned. 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 dot, comma, colon, or semicolon
6. A word longer than six letters
7. A word without the letter *e*

Refer to the table in the [chapter summary](http://eloquentjavascript.net/09_regexp.html#summary_regexp) for help. Test each solution with a few test strings.

// Fill in the regular expressions

verify(/.../,

["my car", "bad cats"],

["camper", "high art"]);

verify(/.../,

["pop culture", "mad props"],

["plop"]);

verify(/.../,

["ferret", "ferry", "ferrari"],

["ferrum", "transfer A"]);

verify(/.../,

["how delicious", "spacious room"],

["ruinous", "consciousness"]);

verify(/.../,

["bad punctuation ."],

["escape the dot"]);

verify(/.../,

["hottentottententen"],

["no", "hotten totten tenten"]);

verify(/.../,

["red platypus", "wobbling nest"],

["earth bed", "learning ape"]);

function verify(regexp, yes, no) {

// Ignore unfinished exercises

if (regexp.source == "...") return;

yes.forEach(function(s) {

if (!regexp.test(s))

console.log("Failure to match '" + s + "'");

});

no.forEach(function(s) {

if (regexp.test(s))

console.log("Unexpected match for '" + s + "'");

});

}

**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.

var text = "'I'm the cook,' he said, 'it's my job.'";

// Change this call.

console.log(text.replace(/A/g, "B"));

// → "I'm the cook," he said, "it's my job."

**Numbers again**

A series of digits can be matched by the simple regular expression /\d+/.

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*.

// Fill in this regular expression.

var number = /^...$/;

// Tests:

["1", "-1", "+15", "1.55", ".5", "5.", "1.3e2", "1E-4",

"1e+12"].forEach(function(s) {

if (!number.test(s))

console.log("Failed to match '" + s + "'");

});

["1a", "+-1", "1.2.3", "1+1", "1e4.5", ".5.", "1f5",

"."].forEach(function(s) {

if (number.test(s))

console.log("Incorrectly accepted '" + s + "'");

});