Regular expressions, or RegExps, are complicated enough to basically be a mini-language inside JavaScript. It takes a while to get the hang of them—far longer than I can cover in this book. Instead of providing a comprehensive reference, I’ll walk you through several examples that should get you started. You can find more examples, tutorials, and reference material in other books and online.

**note** JavaScript regular expressions have a lot of features, but they aren’t as fully featured as Perl-compatible regular expressions and RegExp engines found in other languages. Make sure you’re looking at references and examples specifically for the JavaScript language.
What Are Regular Expressions?

A regular expression describes a pattern. You can search for that pattern in a string. The most basic pattern is a series of alphanumeric characters.

```javascript
var pattern = /match/;
```

Regular expressions sort of look like strings, except they use forward slashes (/) instead of quote marks to delineate the beginning and end. Like strings, you have to escape special characters with backslashes (\). There are more special characters in regular expressions:

```
^ $ . * + ? = ! : | \ / ( ) [ ] { }.
```

Methods

You can use four string methods and two RegExp methods to work with regular expressions. The method examples in this section use the following RegExp and string:

```javascript
var pattern = /match/;
var string = "does this match the pattern?";
```

`string.match(pattern)`

The `match()` method is probably the most common string method. It returns an array of the matched strings. It returns only the first occurrence unless you use the global modifier, which I describe in the “Counting the Occurrences of a Word” section later in this chapter.

```javascript
string.match(pattern);
["match"]
```
Strings vs. Regular Expressions

These examples are pretty lackluster since they use such a simple regular expression. In fact, you would be better off not using regular expressions in these cases because simple string methods are always faster.

```javascript
string.indexOf("match");
10
string.split("match");
["does this ", " the pattern?"]
string.replace("match", "MATCH");
"does this MATCH the pattern?"
```

You’ll learn the real power of regular expressions in the “Examples” section of this chapter.

If the pattern doesn’t match the string, it returns `null`. This is very important because it’s very easy to cause an error by acting on a `null` value like an array.

```javascript
var match = "this won't work".match(pattern);
match[0];
TypeError: match is null
```

`string.search(pattern)`

The `search()` method is similar to `indexOf()`. It returns the position of the match in the string or `-1` if it’s not found.

```javascript
string.search(pattern);
10
```
**string.split(delimiter, limit)**

The delimiter argument in the `split()` method can be a regular expression or just a simple string. It returns an array of strings split on the delimiter argument.

```javascript
string.split(pattern);
["does this ", " the pattern?"]
```

**string.replace(pattern, replacement)**

Like `split()`, the pattern argument can be a regular expression or a string. It returns a new string with pattern replaced by replacement.

```javascript
string.replace(pattern, "MATCH");
"does this MATCH the pattern?"
```

**pattern.exec(string)**

The `exec()` method is almost the opposite of the `match()` method, with string as the argument instead of the object. (There are some important differences, explained in the sidebar at the end of this chapter.) It also returns `null` if the string doesn’t match the regular expression.

```javascript
pattern.exec(string);
["match"]
```

**pattern.test(string)**

The `test()` method is the simplest regular expression method: It simply returns `true` if the string matches the pattern, and it returns `false` otherwise. Use `test()` instead of `exec()` if you need a `true` or `false` value only.

```javascript
pattern.test(string);
true
```
Examples

The following are examples that will get you started using regular expressions. The examples are far from exhaustive, but they cover the concepts and patterns that I use most often.

Match Nearly Anything

Regular expressions use special character combinations to represent patterns in strings.

```
/:.*:/exec("put almost anything you want here");
["put almost anything you want here"]
```

The .* pattern is very common and breaks down like this:

- The period (.) represents any character except for a newline.
- The asterisk (*) signifies zero or more occurrences of the preceding pattern.

Combined, the two characters represent zero or more occurrences of anything but newlines. This is commonly used as a subpattern in a more complex regular expression.

The less “greedy” version of this pattern is .*?, which matches the fewest number of characters. In this example, it would match zero characters. It’s also best used as a subpattern in larger patterns, where it would match the fewest characters between other subpatterns.

Counting the Occurrences of a Word

Regular expressions have the ability to search for multiple occurrences of a pattern. To enable this ability, turn on the global modifier on the regular expression by adding g at the end.
var pattern = /wood/g;
var str = "How much wood could a woodchuck chuck if a woodchuck could chuck wood."
var matches = str.match(pattern);
matches.length;
4

Notice that the pattern matched wood even though it was part of woodchuck.

Another useful modifier is i, which is for case-insensitive matches. You also can combine modifiers.

var caseInsensitivePattern = /word/ig;
var str = "A word, any WORD, regardless of capitalization: WoRd";
var matches = str.match(caseInsensitivePattern);
matches;
["word", "WORD", "WoRd"]

**Counting Words in a String**

The previous pattern matched a specific word, but you can also create a pattern that matches any word by using a character class.

Character classes represent a specific set of characters. The period (.) is a character class that represents the most characters. The character class for characters in words is \\w.

var dummyText = "There are seven words in this string."
var matches = dummyText.match(/\\w+/g);
matches.length;
7
matches;
["There", "are", "seven", "words", "in", "this", "string"]
The plus sign (+) is similar to the asterisk (*) except that it means “one or more.” That’s how a simple pattern like \w+ can match words that are a single character and words that are 15 characters.

Another way to get the same result is splitting the string on spaces. You could just split on a string, but notice that there are three spaces between seven and words.

dummyText.split(" ");
["There", "are", "seven", ",", ",", "words", "in", "this",
⇒ "string."]

The extra spaces create more split points. Instead, you want to split on any occurrence of one or more spaces, using the whitespace class (\s).

dummyText.split(\s+/);
["There", "are", "seven", "words", "in", "this", "string."]

**note** The results using match with \w and split with \s aren’t exactly the same. \w doesn’t match period, but splitting on \s leaves the period at the end of the string.

**Quickly Testing Against Many String Values**

You can include multiple patterns in a regular expression and match a string against any of them. This code is somewhat unwieldy.

```javascript
var input = form.getElementsByTagName("input")[0];
if (input.type === "text" ||
    input.type === "password" ||
    input.type === "email") {
}
```
Instead of repeating `input.type ===` over and over, you can create a regular expression that matches any of the potential values using the alternate symbol (\|, a vertical pipe).

```js
if (/text|password|email/.test(input.type)) {
}
```

**Match a Whole Word**

The character class `\b` matches a *word boundary*, which could be a space, punctuation, or the end of a string.

```
/\btest\b/.exec("This is a test.");
["test"]
/\btest\b/.exec("But not a retest.");
null
/\btest\b/.exec("Still testing.");
null
```

By surrounding a word with `\b` classes, the pattern matches “test” only if it’s by itself, and not as part of another word.

This is useful for testing to see whether an element has a particular class name. Most JavaScript libraries include a function similar to this:

```js
function hasClass(element, className) {
    var pattern = new RegExp("\b" + className + "\b");
    return pattern.test(element.className);
}
```

Here are two important details for this function:

- It creates the regular expression dynamically by passing a string to the `RegExp()` constructor function.
You have to escape the backslashes to use character classes like `\b` in a string passed to new RegExp().

You can use the hasClass() function as follows:

```html
<p class="special">Special paragraph</p>
<p class="extraspecial">Extra special paragraph</p>

hasClass(paragraph1, "special");
true
hasClass(paragraph2, "special");
false
```

Using word boundaries lets you match the word `special` by itself, by not as part of `extraspecial`.

Here are two more useful functions for managing class names on elements:

```javascript
function addClass(element, className) {
    if (!hasClass(element, className)) {
        element.className += " " + className;
    }
}

function removeClass(element, className) {
    var pattern = new RegExp("\b" + className + "\b");
    element.className = element.className.replace(pattern, "");
}
```

**Matching File Extensions**

The following pattern matches `.jpg`, `.jpeg`, `.gif`, and `.png`, but only if they are at the end of the string:
The pattern breaks down as follows:

- Most of the pattern exists inside parentheses, which denotes a subpattern. You’ll see why in the last bullet point.
- The pattern starts with an escaped period (\.). (An unescaped period would match any character.)
- The question mark (?) after the e in jpeg means that it’s optional, or “zero or one occurrence.”
- The alternate subpatterns are separated by the vertical pipe (|).
- The pattern ends with the end anchor ($), so the string has to end with this pattern to match. The end anchor applies to the whole subpattern. Without the parentheses, the anchor would apply only to the g in png.

**Using Character Classes**

You can create your own character classes by listing the applicable characters in brackets ([ ]). This pattern will match any vowel:

```javascript
var vowels = /[aeiou]/;
```

You can also create a character class that matches any character not in the brackets by adding a carat (^) at the beginning.

```javascript
var consonants = /[^aeiou]/;
```

Instead of typing out the whole alphabet when you want to match a letter, you can use ranges:
var allLetters = /[a-z]/;
var capitalLetters = /[A-Z]/;
var numbers = /[0-9]/;
var lettersAndNumbers = /[a-zA-Z0-9]/;

Many classes have shorthand versions. You’ve already seen word (\w) and whitespace (\s) classes. The digit class is \d.

The shorthand classes also have exclusionary versions. To match everything but a word character, use \W. Also, \S is everything but whitespace, and \D is everything but digits.

**Matching a ZIP Code**

A U.S. ZIP code is exactly five digits, which makes for a very simple pattern. But there are still several ways to write an appropriate regular expression.

You could have five characters classes for 0 to 9.

```javascript
/[0-9][0-9][0-9][0-9][0-9]/.exec("12345");
["12345"]
```

It’s a lot simpler to just use the \d shorthand, though.

```javascript
/\d\d\d\d\d/.exec("12345");
["12345"]
```

You can make it even shorter by specifying the repeat count with curly braces ({}).

```javascript
/\d{5}/.exec("12345")
["12345"]
```

One problem with this pattern is that it will match a substring in a longer string. If you’re validating form input, you don’t want to allow the user to accidentally add an extra number.
// Too many digits, still matches first five
/\d{5}/.exec("123456790");
["12345"]

You can use the start anchor (^) and end anchor ($) characters to ensure that the string contains nothing but five digits.

/^\d{5}$/ exec("12345");
["12345"]

Matching IP Addresses

For this example, IP addresses are four numbers between 0 and 999, separated by periods. (Valid IP addresses are more complicated than that, but for this example, that’s our definition.)

Based on the ZIP code example, you could write a pattern like this:

/\d{1,3}\./\d{1,3}\./\d{1,3}\./\d{1,3}/.exec("127.0.0.152");
["127.0.0.152"]

The {1,3} repeat counts specify a minimum of one occurrence and a maximum of three.

You may notice that there’s some unnecessary repetition in that pattern. The \d{1,3}\. portion repeats three times. You could shorten the pattern by creating a subpattern and adding another repeat count.

/\d{1,3}\./\d{1,3}\./\d{1,3}\./\d{1,3}/.exec("127.0.0.152");
["127.0.0.152", "0."]

The subpattern is \d{1,3}\., and it repeats three times before the pattern ends with another number: \d{1,3}.

The array returned by exec() has a second value. Subpatterns are often used to extract a piece of the match. In this case, you don’t care about
the matched subpattern, so you can make the subpattern *noncapturing* by putting `?:` at the beginning.

```
/(?:\d{1,3}\.)\{3}\d{1,3}/.exec("127.0.0.152");
["127.0.0.152"]
```

**Matching HTML Tags**

HTML is too complicated for regular expressions, but here are a few simple examples that might be useful.

Matching a specific tag and its contents is easy.

```javascript
var htmlString = '<p>Some text and <a href="#">a link</a></p>;
/<p>.*</p>/.exec(htmlString);
["<p>Some text and <a href="#">a link</a></p>"]
```

The pattern simply looks for the beginning and ending tags with anything (*) in the middle. Note that you have to escape the forward slash in the ending tag (`</p>`) to keep from ending the regular expression early.

This pattern fails if you have attributes in your HTML tags, but it’s not hard to find those too.

```
/<a\b[^>]*>.*<\/a>/\.exec(htmlString);
["<a href="#">a link</a>"
```

The changes to the pattern are as follows:

- It looks for an `<a>` tag instead of a `<p>`.
- After the tag name, look for a word boundary (`\b`). This ensures you match the specific tag, and not `<article>` by accident.
- Before the opening tag ends, the pattern captures anything that’s not a greater-than symbol (`>`), with this character class: `[^>]`. There can be zero or more occurrences (`*`).
You can make the pattern match any HTML tag by using subpatterns and *back references*. Back references refer to a subpattern match earlier in the string.

Make the tag name generic by replacing `a` with `([A-Z][A-Z0-9]*)`, which is a letter followed by zero or more letters and numbers.

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

The second value in the returned array is the subpattern match. You can refer to that match in the pattern itself with `\1`. (If there were more subpatterns, you could use back references `\2` through `\9`.)

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

Now the pattern can match any HTML tag, its attributes, and its contents.

```
var htmlString = '<h1 id="title">Title Text</h1>';
/<([A-Z][A-Z0-9]*)\b[^>]*>.*</\1>/i.exec(htmlString);
```

But like I said, HTML is too complicated for regular expressions. Consider this example:

```
var pattern = /<([A-Z][A-Z0-9]*)\b[^>]*>(.*)</\1>/i;
var string = "<p>paragraph 1</p> <p>paragraph 2</p>";
string.match(pattern);
```

I changed the pattern slightly, adding parentheses around the greedy catchall pattern (. *). I wanted to extract the content from the first `<p>`,
but instead the pattern matched “paragraph 1</p> <p>paragraph 2.” You could use the less greedy pattern (.*?) to fix this:

```
var pattern = /<([A-Z][A-Z0-9]*)\b[^>]*>(.*?)</\1>/i;
var string = "<p>paragraph 1</p> <p>paragraph 2</p>";
string.match(pattern);
["<p>paragraph 1</p>", "p", "paragraph 1"]
```

**Extracting Parts from IDs and Classes**

You’ve seen subpatterns used for repetition and back references, but they’re usually used to extract a substring from the match. I often use repetitive class names or IDs in HTML to specify something important about my elements.

```html
<a href="#" id="link_1" >Link 1</a>
<a href="#" id="link_2">Link 2</a>
```

Then I use a pattern to match the attribute value and extract the number.

```
var match = /link_(\d+)/.exec(link.id);
match;
["link_1", "1"]
var index = match[1];
```

Remember that exec() and match() return null when the pattern doesn’t match. You can safeguard your code against null values with a conditional expression:

```
var index = (match || [])[1];
```

If match is null, you grab the [1] item from an empty array instead of causing an error by grabbing it from null.
Extracting a Query String Variable

You can extract a single value from a query string with regular expressions.

If `window.location.search` returns this value:

"?param=value&param2=value2"

then you can extract value with this regular expression:

/param=(^[^&]*)/

The subpattern matches all characters except for the ampersand with the `[^&]` character class.

You can also write a generic function to create these patterns dynamically.

```javascript
function getQueryParam(name, queryString) {
  var pattern = new RegExp(name + "=(^[^&]*)");
  var matches = queryString.match(pattern);
}

getQueryParam("param", window.location.search);
"value"
getQueryParam("param2", window.location.search);
"value2"
```

In the previous example, I guarded against null return values with a conditional expression (value || []). You may find the ternary operator (test ? ifTrue : ifFalse) in this function more readable.
Converting Hyphenated Words to camelCase

CSS properties like font-weight must be converted to camelCase (font-Weight) for use in JavaScript. You can do this programmatically with the replace() method and regular expressions.

"margin-left".replace(/-[a-z]/ig, function(match) {
    return match.substr(1).toUpperCase();
});
"marginLeft"

The pattern passed into replace() finds a hyphen followed by a letter. It has the case insensitivity and global modifiers in case there are capital letters or it’s a word with multiple hyphens.

The second argument to replace() can be a function. The first argument of that function is the matched string. In this case, match is “-l”. The function extracts the second character with substr() and calls toUpperCase(). The value returned by the function replaces the matched string.

You can reverse the process with a similar pattern and function.

"marginLeft".replace(/[a-z][A-Z]/g, function(match) {
    return match.charAt(0) + "-" + match.charAt(1).toLowerCase();
});
"margin-left"

This time, the pattern finds a lowercase letter followed by an uppercase letter, "nL" in this case. The replacement function returns the first letter, n, a hyphen, and the lowercase second letter.
String Templates

The `replace()` method is very powerful. One of the cleverest uses I’ve seen lets you create HTML templates and replace keywords with data from various objects.

Say you wanted to create the following HTML many times dynamically based on different data:

```
<h3>Person's Name</h3>
<p>Age: person's age<br />
Favorite Browser: person's favorite browser</p>
```

Start by creating a template string with keywords surrounded by curly brackets, as shown here:

```
var template = "<h3>{name}</h3><p>Age: {age}<br />Favorite Browser: {browser}</p>";
```

The keywords should match the property names of your data objects.

```
var lenny = {
    name : "Lenny",
    age : 26,
    browser : "Safari"
};
```

```
var julie = {
    name : "Julie",
    age : 24,
    browser : "Firefox"
};
```

The pattern to match the keywords in curly brackets is ugly but also pretty simple.
\/{\{([^}]\}+)\}\}/g

Part of the ugliness comes from having to escape each bracket, since they're used for repeat count qualifiers. The pattern matches an opening bracket, a subpattern of one or more characters that aren't closing brackets ([^\}]\}+), and a closing bracket.

The templating function looks like this:

```javascript
function templatize(template, data) {
  var key = \/{\{([^}]\}+)\}\}/g;
  return template.replace(key, function(match, subpattern) {
    return data[subpattern];
  });
}
```

templatize(template, lenny);
<h3>Lenny</h3><p>Age: 26<br />Favorite Browser: Safari</p>
templatize(template, julie);
<h3>Julie</h3><p>Age: 24<br />Favorite Browser: Firefox</p>

Subpattern matches are passed to the replacement function as arguments, so you can use the match as the property name for the data object. The first subpattern match is *name*, so the value returned is `data["name"]`.

I much prefer creating complex HTML strings with a template instead of concatenation. It's a lot easier to reuse the `templatize()` function than to write all your HTML in JavaScript like this:

```javascript
return "<h3>" + data.name + "</h3><p>Age: " + data.age + "<br />Favorite Browser: " + data.browser + "</p>";
```
The Difference Between \texttt{match()} and \texttt{exec()}

You can’t use \texttt{match()} for global searches with subpatterns because \texttt{match()} ignores subpatterns when you use the \texttt{g} modifier. It would be too confusing to tell which results are full pattern matches and which are subpattern matches.

\begin{verbatim}
var string = "key_1 key_2 key_3 key_4";
var pattern = /key_(\d+)/g;

string.match(pattern);
["key_1", "key_2", "key_3", "key_4"]
\end{verbatim}

To search globally and use subpatterns, you have to loop over the result of \texttt{exec()}. Each time you execute \texttt{exec()} on a RegExp, it remembers the position of the last match. After it finds the last match, \texttt{exec()} returns null, and the loop ends.

\begin{verbatim}
var match;
while (match = pattern.exec(string)) {
    console.log(match);
    console.log(pattern.lastIndex);
}
["key_1", "1"]
5
["key_2", "2"]
11
["key_3", "3"]
17
["key_4", "4"]
23
\end{verbatim}