Riverside JS Workshop
JavaScript Skills Ladder, Lesson 2.10

Strings and String Methods

Lesson Objectives

When you complete this lesson, you will be able to:

- *build a string search application.*
- *explore several of the different methods we have for manipulating strings in JavaScript.*
- *use methods and chain those methods to improve your searches.*
- *use the substring() and split() methods.*
- *use regular expressions to create a pattern to match in some text.*

It's time for a well-deserved break from the To-Do List Application. In this lesson, we'll build a string search application and, in the process, explore several of the different methods we have for manipulating strings in JavaScript. You've used strings many times in this course, but we haven't done much with them except to put them into web pages. There is a lot more you can do with strings, as you'll soon discover.

String Basics

You already know that to create a string in JavaScript, you write some text in quotes, like this:

OBSERVE:

```
var myString = "I'm a string!";
```

It's fine for a string to contain a single quote, as long as you're using double quotes to delimit the string, but if you need a double quote in a string, you need to *escape* it, like this:

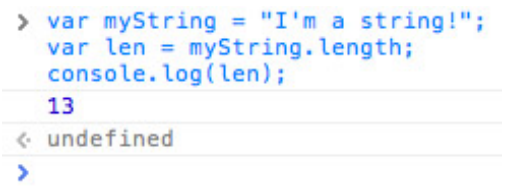
OBSERVE:

```
var myQuoteString = "He said, \"Give me the ice cream!\" but I
didn't.";
```

Strings in JavaScript are a bit special. When you write a string, you're actually creating a *String object*. Just like other objects, String objects have methods and properties. You already know about one property, *length*. Type this into your browser's JavaScript console:

CODE TO TYPE:

```
var myString = "I'm a string!";
var len = myString.length;
console.log(len);
```



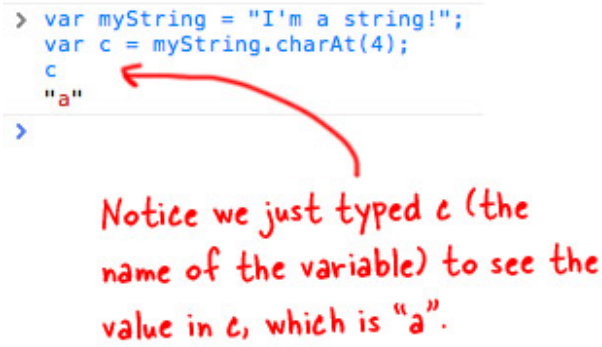
Try working through a few other strings for a little more practice.

Note When you want to see the value of a variable in the JavaScript console, you don't actually have to use `console.log()`; you can just type the name of the variable. For example, if you want the value of `len`, you can just type `len` at the console prompt.

Let's try a String method, `charAt()`:

CODE TO TYPE:

```
var myString = "I'm a string!";
var c = myString.charAt(4);
c
```



If "a" is the fifth character in the string, how did `charAt()` get "a"? It retrieved the character at position (also called **index**) 4. Just like arrays, strings start with position 0, so if you count 0, 1, 2, 3, 4 characters (including the space!), you'll find "a" in position 4. But just because you can

access a character in a String using a position, or index, don't confuse it with an Array because they are two entirely different objects.

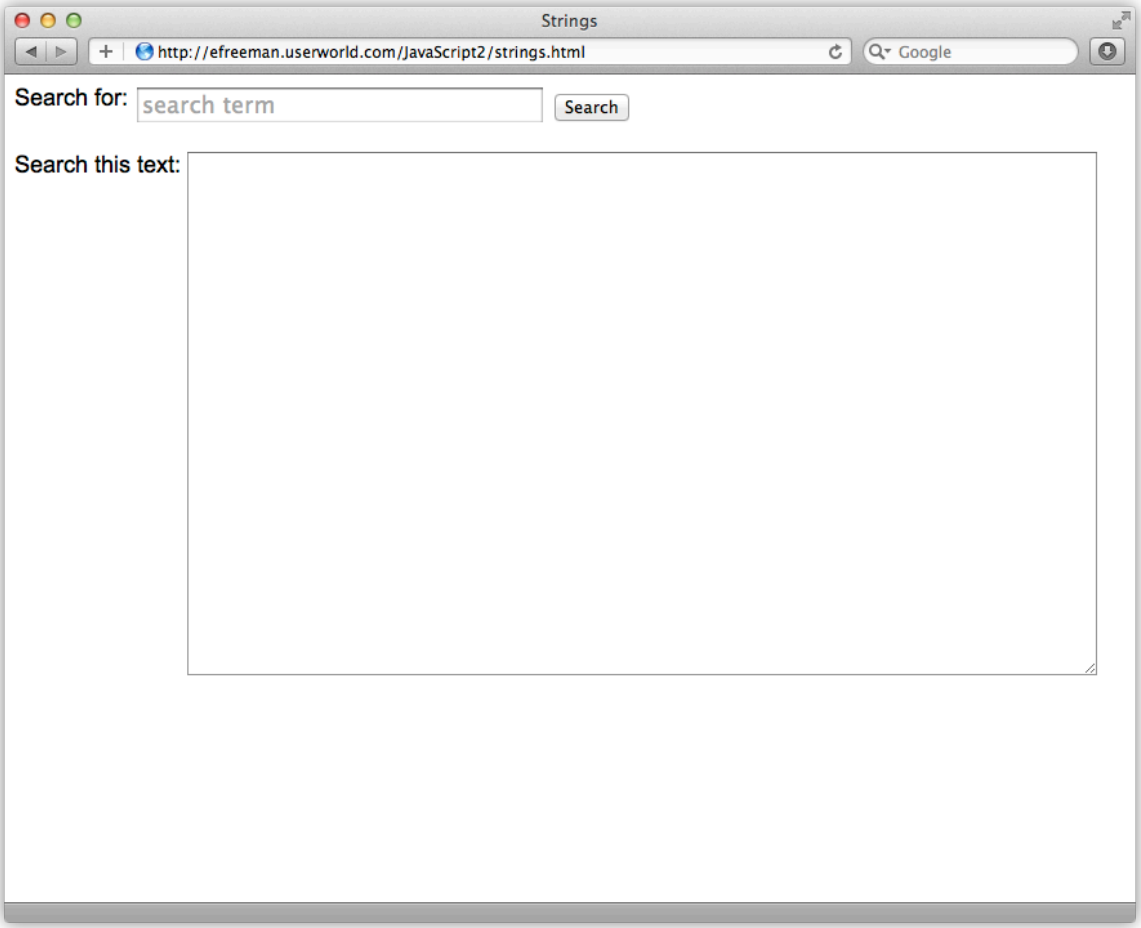
Basic String Comparison and Searching

Now that you know some string basics, let's start writing our string search application. We'll start with some basic HTML. Create this file:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Strings</title>
  <meta charset="utf-8">
  <script src="strings.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    textarea {
      width: 700px;
      height: 400px;
    }
    label {
      vertical-align: top;
    }
  </style>
</head>
<body>
   <form>
     <label for="searchTerm">Search for: </label>
     <input type="text" id="searchTerm" size="35"
            placeholder="search term">
     <input type="button" id="searchButton" value="Search"><br><br>
     <label for="textToSearch">Search this text:</label>
     <textarea id="textToSearch"></textarea>
   </form>
</body>
</html>
```

Save the file as *strings.html* in your work folder, and then load the page into a browser. You see this:



Nothing works yet because we haven't written the JavaScript. The plan is that when you click the Search button, we'll begin the search process and search for the string you enter in the top search area within the string in the textarea at the bottom. Notice that we're linking to the file *strings.js* from the HTML. That's where we'll add the JavaScript to make this work. Let's do that now.

Create a new file and enter this JavaScript:

CODE TO TYPE:

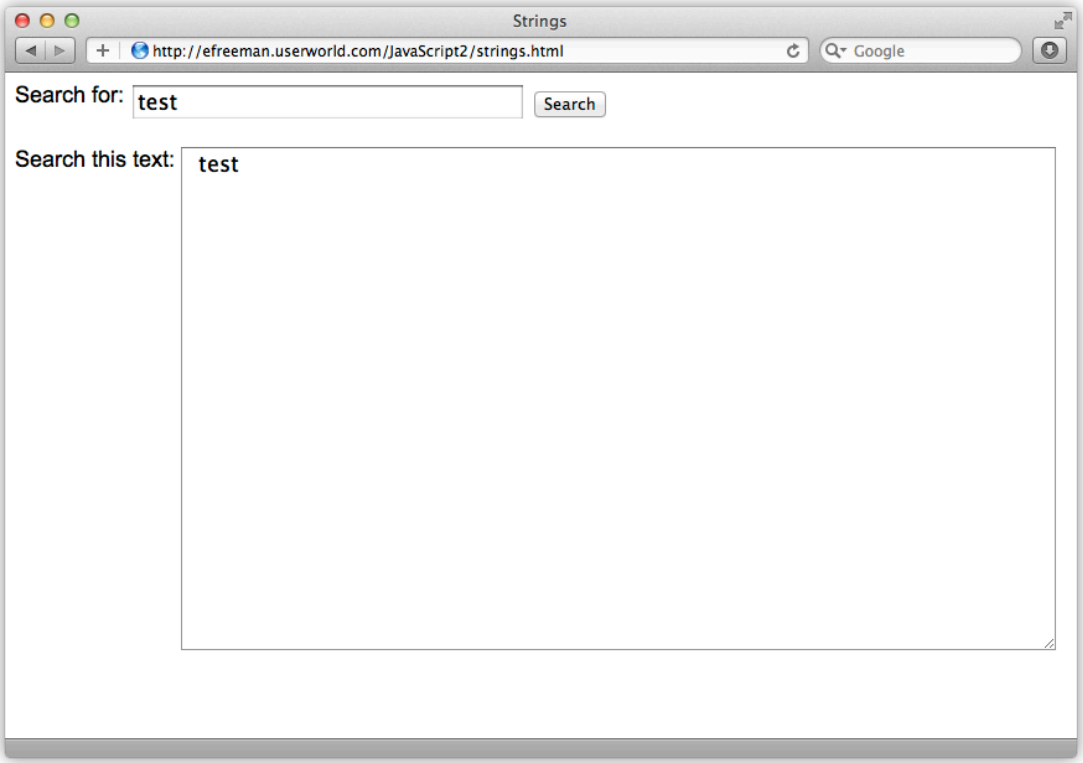
```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
```

```
    var textToSearch = document.getElementById("textToSearch").value;
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }
    if (searchTerm == textToSearch) {
        alert("Found 1 instance of " + searchTerm);
    }
    else {
        alert("No instance of " + searchTerm + " found!");
    }
}
```

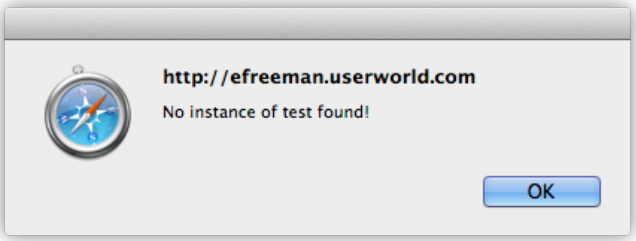
Save this file as *strings.js* in your work folder, and open or reload *strings.html* in your browser. Try entering a search string and some text to search. You'll probably notice two things right away:

- If either of the strings you enter has extra white space at the beginning or the end, you might find your search doesn't work, even if it appears that the text in both boxes is exactly the same.
- Both strings must be *exactly* the same in order for the search to work, which isn't particularly useful.

Don't worry. We'll fix both of these problems. First things first -- try entering "test" in the search area, and " test " in the text area:



These strings won't match when you click the Search button:



The code that tests to see if the two strings are the same:

OBSERVE:

```
if (searchTerm == textToSearch) {
    alert("Found 1 instance of " + searchTerm);
}
```

The == operator checks to see if they are *exactly* the same, spaces included. When comparing two strings using ==, JavaScript compares the two strings, character by character, to see if they

are the same, and that includes any spaces in the quotes. "test" does not equal " test ", so the message we get is correct.

The `trim()` method removes leading and trailing spaces from a string. This function is handy. Adding extra spaces is a common error and typically people don't *really* want those extra spaces when they are comparing strings. (In some situations they might, however, so you'll need to take this on a case-by-case basis).

Modify *strings.js* as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }
    if (searchTerm == textToSearch) {
        alert("Found 1 instance of " + searchTerm);
    }
    else {
        alert("No instance of " + searchTerm + " found!");
    }
}
```

Save the changes to *strings.js*, and open or reload *strings.html* in the browser. Now two strings will match if they have the same letters, even if you use leading or trailing spaces. Try it! Try "test" and " test " again.

The built-in trim function is only available in recent versions of most browsers: Firefox 3.5+, Safari 5+, IE9+, Chrome 5+, and Opera 10.5+. For browsers that don't support the built-in function, you can substitute your own implementation:

OBSERVE:

```
function trim(str) {
    return str.replace(/^\s+|\s+$/g,"");
}
```

And then instead of calling `myString.trim()`, you'd write `trim(myString)`.

Improving the Search

So far, our search is rather underwhelming, but there are lots of things we can do to improve it. Let's start by using two additional String methods, `indexOf` and `toUpperCase()`.

The `indexOf()` method takes a string to find in the string it's called on, and returns the position of the first instance it finds. So, if you enter the following code into the console,

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var pos = myString.indexOf("scream");
pos
```

...you'll get the value *2* in the variable `pos`, because the first instance of "scream" starts at position 2 in the variable `myString`. The method, `indexOf()`, can also take an optional argument, a starting position, so that you can start looking for a string at that position. If you don't supply this argument, the default is to start looking at position 0, the beginning of the string.

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var pos = myString.indexOf("scream", 3);
pos
```

Now you'll get *14*, the position of the first instance of "scream", starting at or after position 3.

If `indexOf()` doesn't find any instances of the string you pass it, then it returns -1.

The function `toUpperCase()` converts the characters in a string to upper case.

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var myStringUpper = myString.toUpperCase();
myStringUpper
```

Our code produces the string "I SCREAM, YOU SCREAM, WE ALL SCREAM FOR ICE CREAM", in the variable `myStringUpper`. There is an analogous `toLowerCase()` method also. Notice that these methods do not change the original string, `myString`, rather, the methods return a *new* string that you can store in a different variable if you want.

Let's use these two methods to improve our search. Modify *strings.js* as shown:

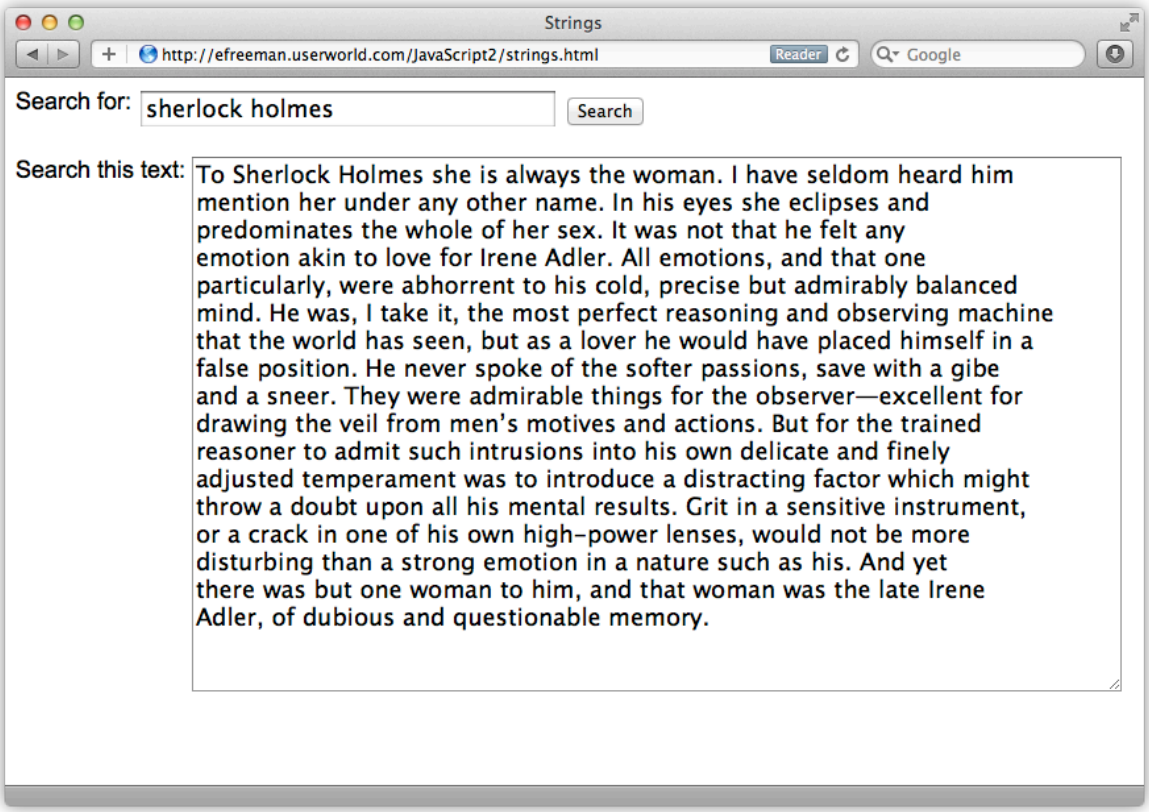
CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }
    if (searchTerm == textToSearch) {
        alert("Found 1 instance of " + searchTerm);
    }
    else {
        alert("No instance of " + searchTerm + " found!");
    }

    var pos = 0;
    var count = 0;
    while (pos >= 0) {
        pos =
textToSearch.toUpperCase().indexOf(searchTerm.toUpperCase(), pos);
        if (pos >= 0) {
            count++;
            pos++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm);
}
```

Save the changes to *strings.js*, and open or reload *strings.html* in the browser. Try a few test strings. Make sure you try:

- lower and upper case words
- words with spaces in them
- words with multiple instances in the text you're searching

We pasted in some text from The Adventures of Sherlock Holmes, and tried some searches on the first paragraph from the book.



How many times does the word "and" appear in the paragraph? Here's what you get if you try:

OBSERVE:

```
var pos = 0;
var count = 0;
while (pos >= 0) {
    pos = textToSearch.toUpperCase().indexOf(searchTerm.toUpperCase(),
pos);
    if (pos >= 0) {
        count++;
```

```
        pos++;
    }
}
alert("Found " + count + " instances of " + searchTerm);
```

Let's break it down. We use a loop so we can find *all* instances of the `searchTerm` string that appear in the `textToSearch` string, and keep track of how many we find using the `count` variable. We use the `pos` variable to keep track of the position where we find each instance of `searchTerm`. As long as `pos` is greater than or equal to 0 we know we've found another instance (remember, `indexOf()` returns -1 when it can't find an instance of the string for which you're searching). We convert both strings to upper case with `toUpperCase()` so that we can find "and" as well as "AND" or "And" (or even "aND"). We use `indexOf()` with the second, optional, argument, `pos`, so that we can start searching at a position that is *after* the position where we found the previous instance (so we don't keep finding the same instance over and over). As long as we keep finding a new instance of the `searchTerm`, we increment the position, and we increment the count. When pos is -1, the loop stops because we've found all the instances of searchTerm there are to find in `textToSearch`.

Chaining

We used a JavaScript technique in this code called *chaining*. Chaining is combining multiple method calls together in one line of code. For instance, if you write:

OBSERVE:

```
var myString = "This is the text we're searching to find the word
'and'.";
var anotherString = "AND";
var pos = myString.toUpperCase().indexOf(anotherString);
```

..it's the same as if you wrote:

OBSERVE:

```
var myString = "This is the text we're searching to find the word
'and'.";
var anotherString = "AND";
var myStringUpper = myString.toUpperCase()
var pos = myStringUpper.indexOf(anotherString);
```

By chaining expressions together with the "dot notation," you eliminate the intermediate variable you'd create if you used two statements instead. You can do this when you know that the result of the first method will yield a value you can use for the second method. In this case,

`myString.toUpperCase()` returns an upper-case string, then we can call the method `indexOf()` on that string.

The Substring() and Split() Methods

Two other useful String methods you'll run into fairly often are `substring()` and `split()`.

`substring()` creates a string from a longer string, like this:

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var myStringSub = myString.substring(0, 8);
myStringSub
```

The value of `myStringSub` is "I scream", a string made from the characters at positions 0-7 of `myString`. Notice that the character at position 8 is *not* included. `substring()` takes two values, `from` and `to`: from is the position of the first character you want included in the substring, and to is *one greater* than the position of the last character you want included in the substring.

The `split()` method is pretty handy. It splits a string into parts using a character as the divider for the split. So, let's say we want to split the text we entered in our string search application into words. We can split the input string using the " " (space) character. The result of `split()` is an array, so in this case, we'd get an array of all the words in the text. Let's modify our string search application a bit to count the number of words, and then search for a word by comparing the search text to each word we found in the text we searched. Modify *strings.js*:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
```

```
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }

    var pos = 0;
    var count = 0;
    while (pos >= 0) {
        pos =
textToSearch.toUpperCase().indexOf(searchTerm.toUpperCase(), pos);
        if (pos >= 0) {
            count++;
            pos++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm);

    var results = textToSearch.split(" ");
    var count = 0;
    for (var i = 0; i < results.length; i++) {
        if (searchTerm.toUpperCase() == results[i].toUpperCase()) {
            count++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm + " out of
a total of " + results.length + " words!");
}
```

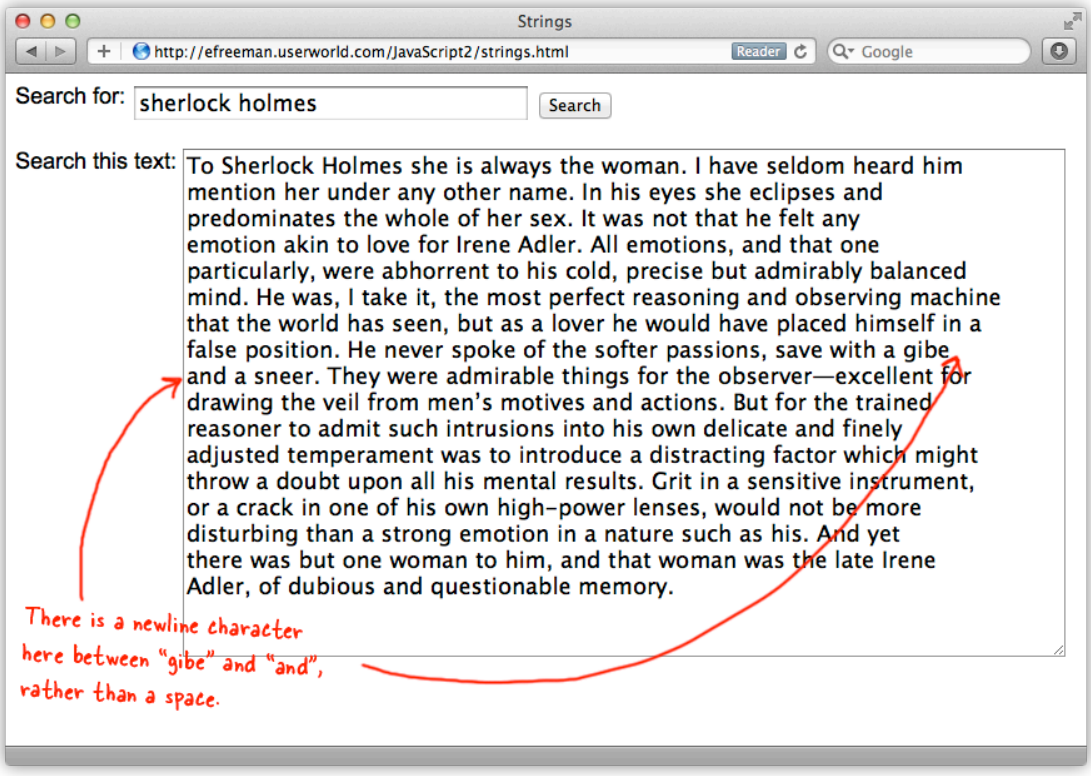
Save the changes to *strings.js*, and open or reload *strings.html* in your browser. Try searching for a string. Now you'll see an alert that displays the number of times the string you searched for was found, as well as how many total words were found:

OBSERVE:

```
var results = textToSearch.split(" ");
    var count = 0;
    for (var i = 0; i < results.length; i++) {
        if (searchTerm.toUpperCase() == results[i].toUpperCase()) {
            count++;
        }
    }
```

First, we split the `textToSearch` into words using `split()`, and stored the results in an array named results. Then, we iterate over the entire array, and compare each word in the array with the word we're searching for, and keep track of how many times we found it.

If you're using the text from the Sherlock Holmes example (above), try the word "and," and you might see that "and" appears 8 times. If you tried "and" earlier (when we were using `indexOf()` rather than `split()`) you probably saw that it was found 9 times in the text. So why 8 this time? If you typed in the text to search on and pressed the "Enter" key on your keyboard between lines (rather than continuing to type and having the lines wrap around), then the words at the end of each line and the beginning of the next line are not separated by a space. Rather, they are separated by a `newline` character. So when you use `split()`, and you split the words up into an array using space (" ") as the split character, these words (the ones at the end of line/beginning of next line) aren't actually split up. So if "and" appears at the end or beginning of a line, it will not appear in the array as "and," but rather as "and" concatenated with another word. In my example, I pressed return between "gibe" and "and" so one of the words in the array was "gibe(newline)and." That word didn't match "and" and that's why I got 8 as the result rather than 9.



Typically, `split()` works best when you know for sure that a given character is used to separate text. For instance, in a CSV file, the "," character is used to delimit the columns in the file. It's also used to split smaller pieces of text, for example, a "firstname lastname" entry could be split into "firstname" and "lastname" using `split()`.

Regular Expressions

So far, we've been searching for exact matches for the search terms we've tried. For instance, we've tried searching for "and" to see how many times the word "and" appears in the text you enter to be searched. But what if you want to find, say, all the phone numbers in a bit of text, even if there is more than one and they are different?

For a task like that you need *Regular Expressions*. Regular Expressions are a powerful way to express patterns to match. You'll find Regular Expressions used frequently whenever text needs to be matched to a pattern. For instance, you can use Regular Expressions to verify that the pattern of text a user enters for a phone number in a form really does look like a phone number—or an email address. There are many uses of Regular Expressions. You'll see them in other programming languages, as well as in command line commands or shell scripts.

In JavaScript, there are a few ways to use Regular Expressions. We'll talk about one of these: `match()`.

Let's update our code to use a Regular Expression and then we'll come back to talk more about how Regular Expressions work. They're a bit mysterious at first, so don't worry if it takes you a while to get used to them. Modify *strings.js*:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }

    var results = textToSearch.split(" ");
    var count = 0;
    for (var i = 0; i < results.length; i++) {
```

```
        if (searchTerm.toUpperCase() == results[i].toUpperCase()) {
            count++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm + " out of
a total of " + results.length + " words!");
```

```
    var re = new RegExp(searchTerm, "ig");
    var results = textToSearch.match(re);
    if (results == null) {
        alert("No match found");
    }
    else {
        alert("Found " + results.length + " instances of " +
searchTerm);
        // Show the matches in the page
        showResults(results);
    }

}
function clearResultsList(ul) {
    while (ul.firstChild) {
        ul.removeChild(ul.firstChild);
    }
}

function showResults(results) {
    var ul = document.getElementById("matchResultsList");
    clearResultsList(ul);
    var frag = document.createDocumentFragment();
    for (var i = 0; i < results.length; i++) {
        var li = document.createElement("li");
        li.innerHTML = results[i];
        frag.appendChild(li);
    }
    ul.appendChild(frag);
}
```

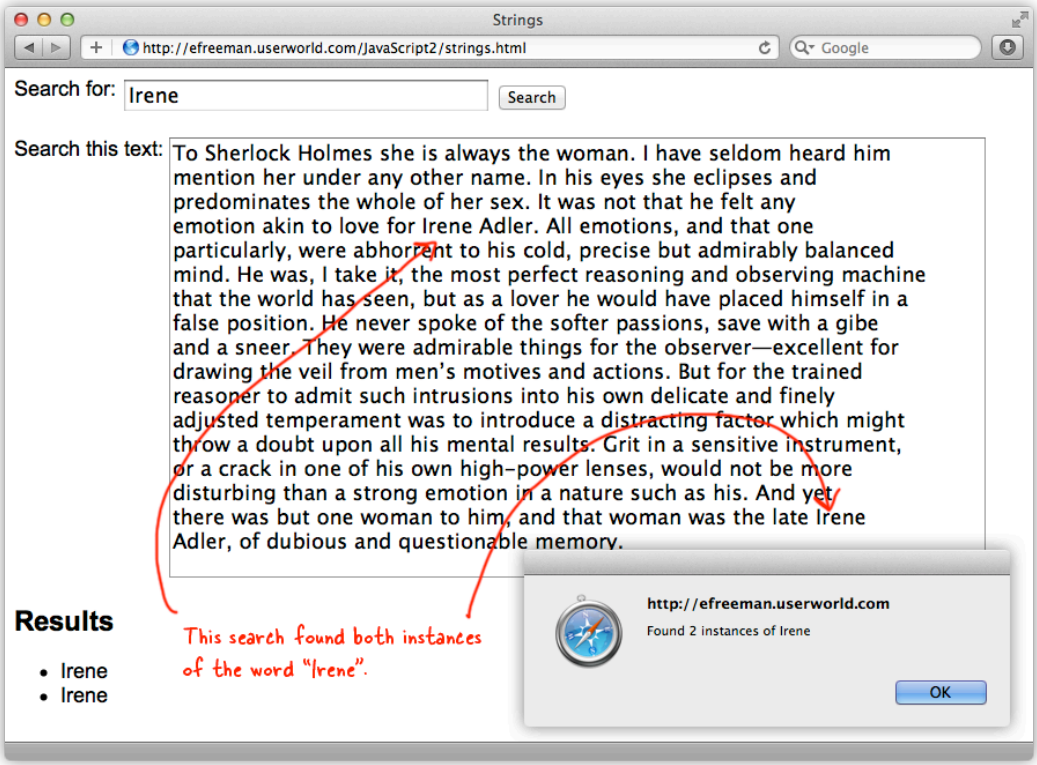
Save your changes. Before you try it though, we need to update *strings.html*, because we're going to update the HTML page with the results of our search match. We're just creating a list of matching words in the `<ul>` element with the id `matchResultsList`, so we need to add this to the page:

CODE TO TYPE:

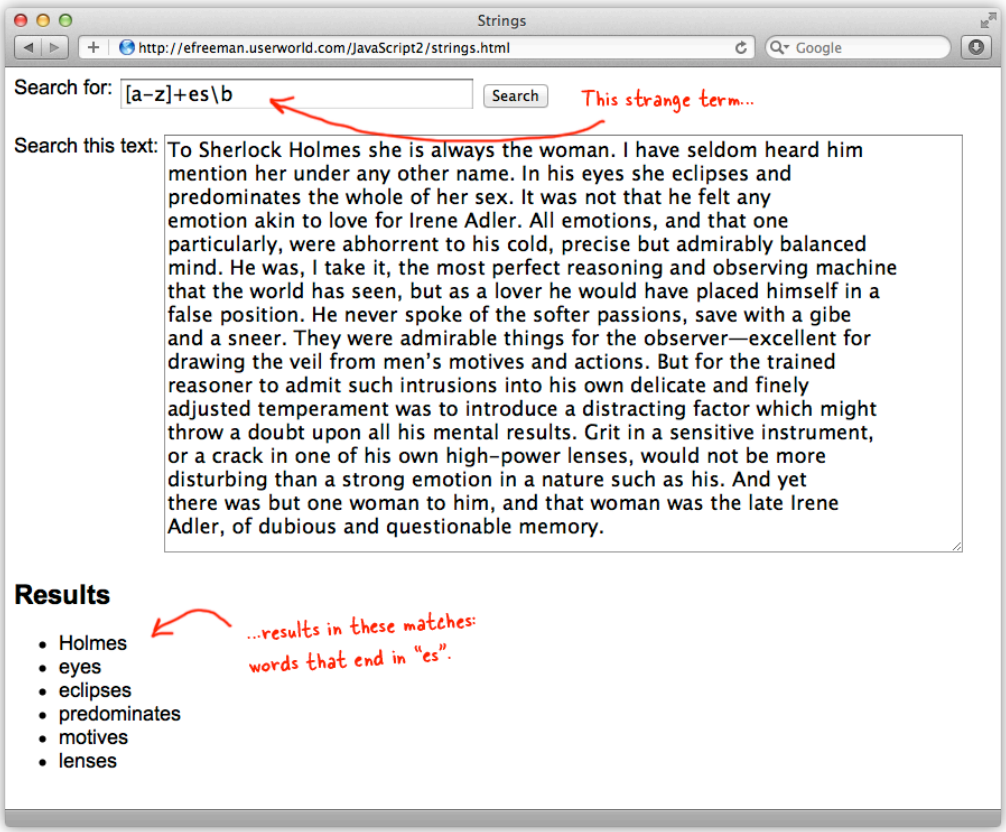
```
<!doctype html>
<html>
<head>
<title>Strings</title>
  <meta charset="utf-8">
```

```
  <script src="strings.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    textarea {
      width: 700px;
      height: 400px;
    }
    label {
      vertical-align: top;
    }
  </style>
</head>
<body>
   <form>
     <label for="searchTerm">Search for: </label>
     <input type="text" id="searchTerm" size="35"
            placeholder="search term">
     <input type="button" id="searchButton" value="Search"><br><br>
     <label for="textToSearch">Search this text:</label>
     <textarea id="textToSearch"></textarea>
   </form>
   <div>
     <h2>Results</h2>
     <ul id="matchResultsList">
     </ul>
   </div>
</body>
</html>
```

Save the changes to *strings.html*, and open or reload the page in your browser. Try searching for a string. For instance, if you enter the text from Sherlock Holmes again, search for "Irene". You'll find two matches. First, you'll see an alert that tells you how many matches you have, and then you'll see those matches in the results list in the page.



So what? We could do that before with `indexOf()`. Okay, try entering `[a-z]+es\b` in the Search for field:



Now you see a list of words that end in "es" in your page. That's pretty cool, right? But you're probably wondering, what on earth is `[a-z]+es\b`? It's a Regular Expression.

Regular Expressions Create a Pattern

Regular Expressions are used to create a pattern to match in some text. The simplest regular expression you can create is an exact word match, like we did above with "Irene." With this kind of regular expression, we just match the pattern "Irene" letter for letter. It works likes `indexOf()`, except that the method we used here, `match()`, returns an array of results, rather than a position.

But you can also provide more complex regular expressions, ones that will match multiple strings. That's what we did with the regular expression "`[a-z]+es\b`", which matches all words that end in "es".

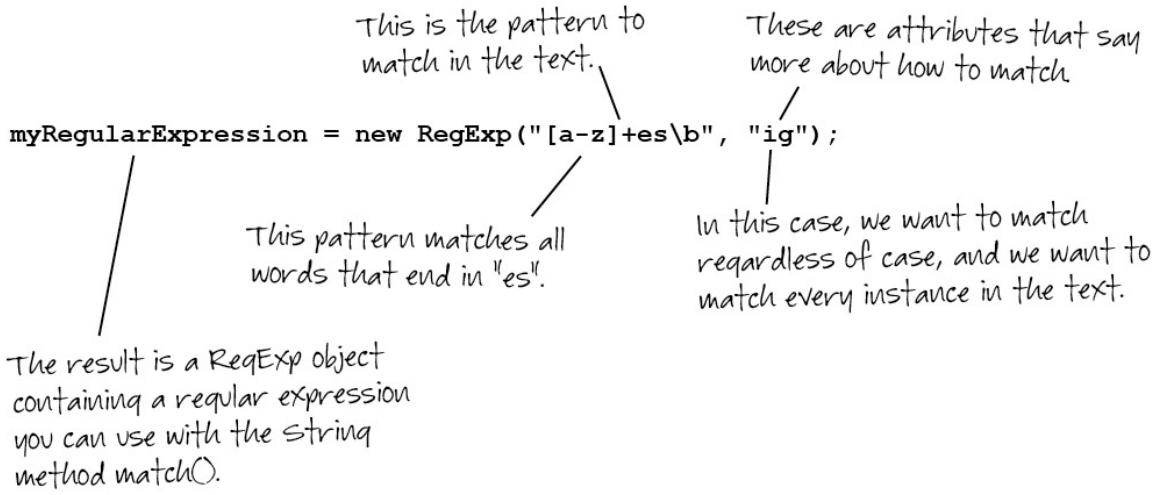
In our code, we create a Regular Expression object, using the `RegExp()` constructor. We pass in the `searchTerm`, which is a *pattern*—that is, the text you typed, like "Irene" or "[a-z]+es\b"—and a second argument, "ig." That second argument is a list of *attributes*. There are a few modifiers you can use; "i" and "g" are common. "i" says to ignore the case (so we'd match

"Irene" to "IRENE" or "irene" or "irENe" or any combo like that), and "g" says to "globally" match every instance in the text, rather than just the first one:

OBSERVE:

```
var re = new RegExp(searchTerm, "ig");
```

This *RegExp* object can then be used to match strings in some text. The pattern can be exact, like "Irene", or it can be set up to match a variety of different words, like all words that end in "es", which is what the expression "`[a-z]+es\b`" does:



OBSERVE:

```
var re = new RegExp(searchTerm, "ig");
var results = textToSearch.match(re);
```

To match the pattern to some text, we use the String method, `match()`. `match()` takes a regular expression re and matches the pattern to the string, in this case `textToSearch`, that we called `match()` on. In the example above, `textToSearch` contains the text from Sherlock Holmes. We match that against the pattern "`[a-z]+es\b`" with the attributes "ig" (that is, find all instances in the text, regardless of their case). The result is an array, which we put into a variable, results.

Once we have the results, we can get the `length` of the array to count how many matches there were, or we can display each result in the page like we have:

OBSERVE:

```
var re = new RegExp(searchTerm, "ig");
var results = textToSearch.match(re);
if (results == null) {
    alert("No match found");
}
else {
    alert("Found " + results.length + " instances of " + searchTerm);
    // Show the matches in the page
    showResults(results);
}
```

We create a function `showResults()` that takes the results array and displays the results in the page. We also create a function `clearResultsList()` to clear the results list each time you submit a new search so you get new results each time. All of the code in `showResults()` and `clearResultsList()` is probably familiar to you.

Regular Expression Patterns

Let's take a closer look at the *attributes* and *patterns* you can use to create a *RegExp* object. Remember that in our search application, our Regular Expression object has the attributes "ig," so for all of these examples, we'll match regardless of case, and we'll search over all the instances in the text. If you want to, try removing one or both of these attributes to see the difference. Experiment!

Matching Characters in a Range [ ]

To match a character in a range, you use `[ ]` to specify the range. For instance, `[a-z]` says to match one letter between a and z; `[0-9]` will match one digit between 0 and 9. Try this. Create some numbers in the text to search, varying the digits. For example, you could use 12, 333, 5985, 2, and so on. Now write a pattern that will match only digits 1, 2 and 3, like this: [1-3]. Enter the pattern in the search field. You should see all the digits you typed that match 0-3, but notice, you're matching only single digits.

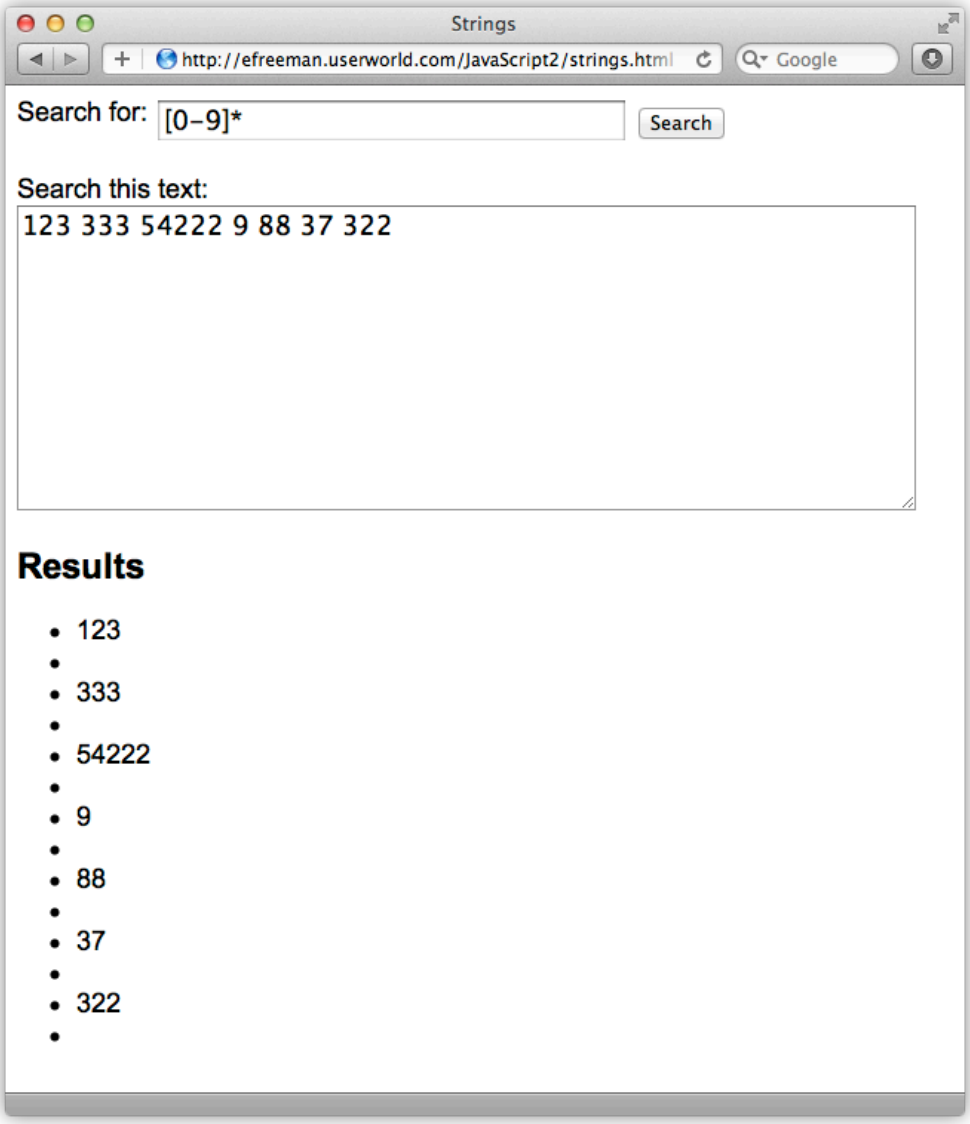
Matching Multiple Characters with *, +, and { }

SO, what if you want to match more than one character? You can use `*` to match any number of characters, including zero. Use + to match one or more characters, and `{ }` to match a specific number of characters. Using your search application, try matching these patterns to the same numbers you entered before:

OBSERVE:

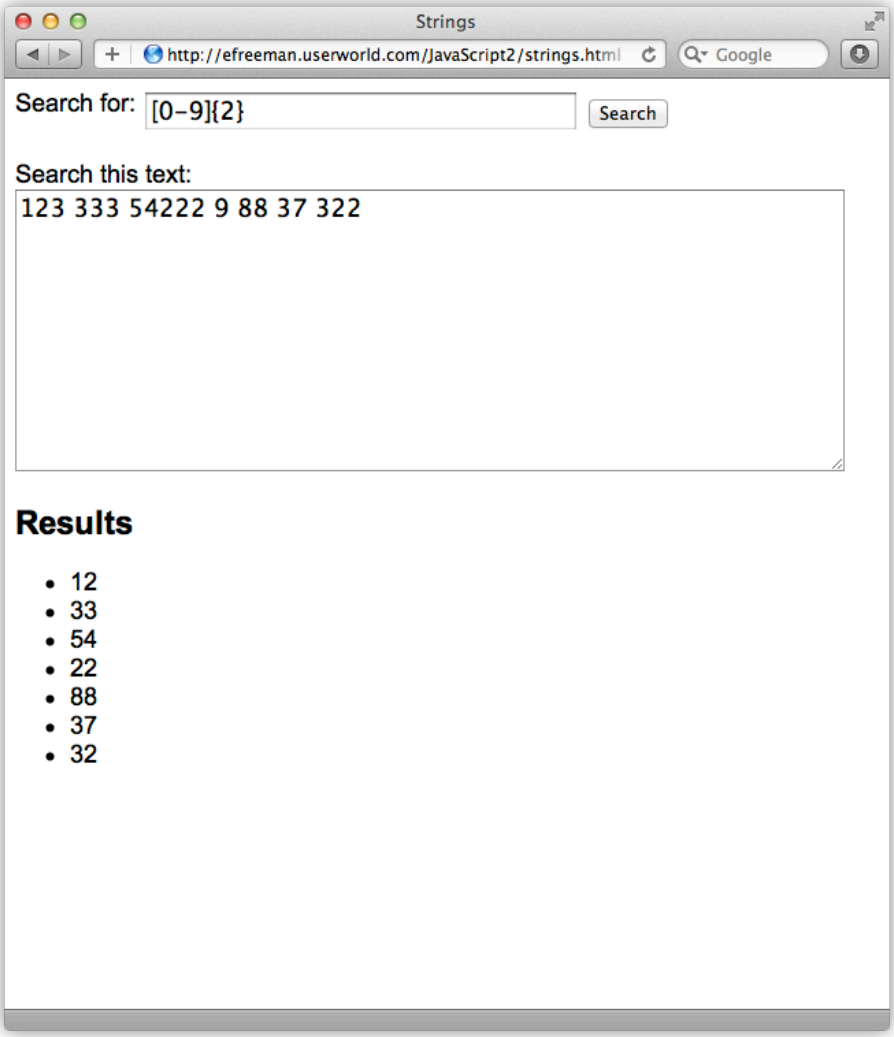
```
[0-9]*
[0-9]+
[0-9]{2}
```

`[0-9]*` says to match any number of characters (or none) as long as they are between 0 and 9. Your results will show all the numbers you typed, as well as matches of no numbers at all for the spaces in between the numbers. Why? Because the * matches *zero or more* characters. So for the first number, say 123, it will match all those digits. Then, for the space between 123 and the second number, say, 333, it will match zero digits. It will include that in the search results, and go on until it reaches the end of the numbers you input.



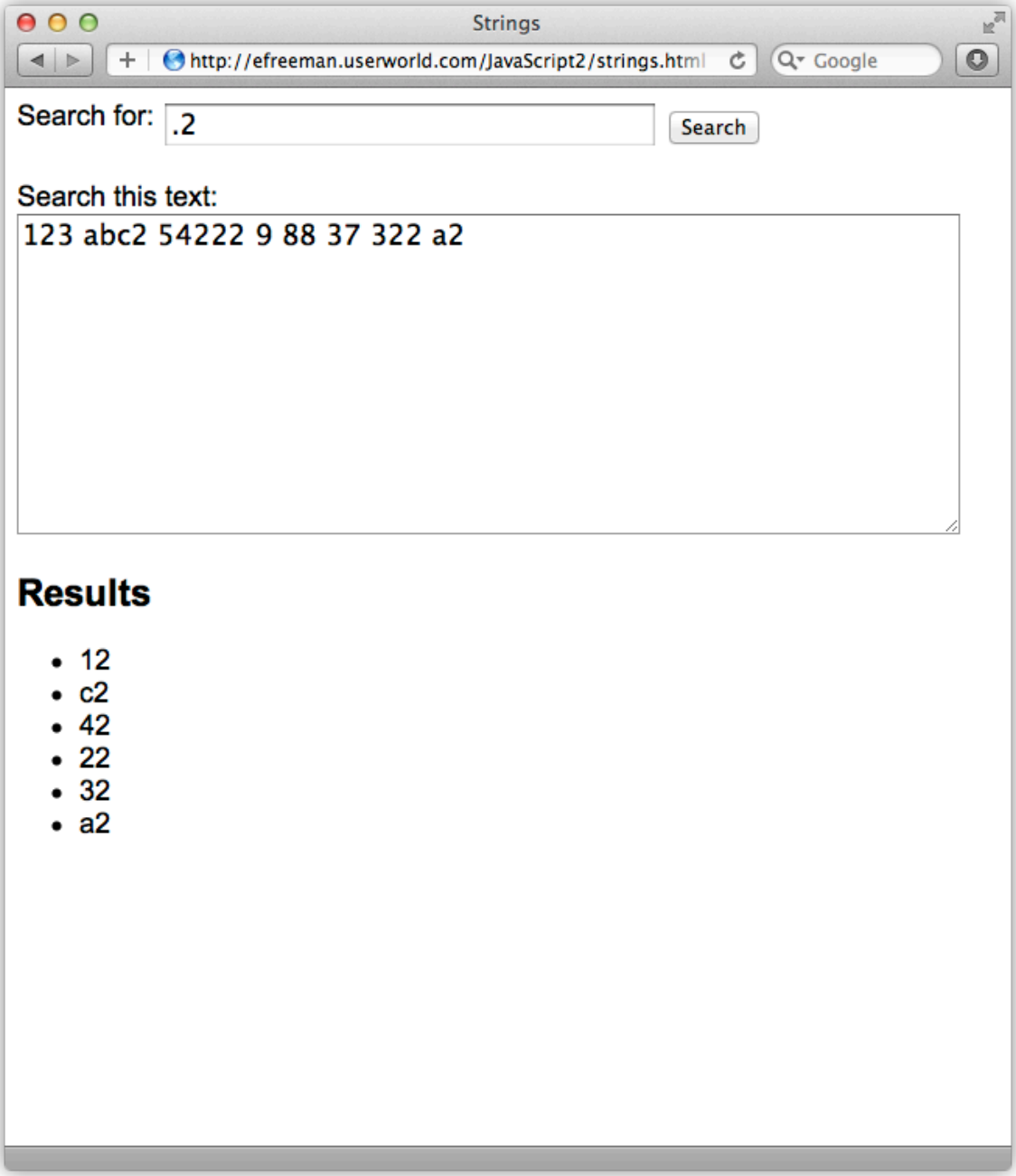
`[0-9]+` says match one or more characters as long as they are between 0 and 9. Compare your results to `[0-9]*`. You'll see that your results are limited to the numbers you typed only. No "zero" results for the spaces in between, right?

`[0-9]{2}` matches 2 numbers precisely. Notice that it matches the sequences of two numbers only once, so if a number has been matched (like 12 in 123) the same number won't be matched again for a different sequence of two numbers (like 23, where the 2 is from the same number, 123).



Matching Characters at a Specific Position

What if you want to match one or more characters at a specific position? Try entering some letters next to a number, like I did here:



`.`2 matches *any non-white-space character* next to the number 2. Try `..`2; do you see three-character results? Now you're matching *two* non-white-space characters next to the number 2.

You can match characters or sequences of characters at the end of a string. Try entering: "I scream, you scream, we all scream for ice cream" in the text to search (don't put a period at the end of the sentence). Now search for:

OBSERVE:

```
cream$
scream$
```

`cream$` matches one "cream" in that sentence, the one at the very end, because `$` says match the pattern at the end of the string.

`scream$` doesn't match with any results, even though "scream" appears three times in the sentence, because "scream" does not appear at the end of the string we're searching.

You can also match on a word boundary (rather than the whole string boundary, like we just did). Using the same sentence, try it:

OBSERVE:

```
cream\b
```

Now you see *four* results. Why? Because the string "cream" appears four times at the end of a word: "scream" three times, and "ice cream" one time, and in each case, "cream" is at the end of the word. The `\b` characters mean "match a word boundary" and because we put `\b` at the end of "cream", we're saying to match the end of the word. Try this instead:

OBSERVE:

```
rea\b
```

You see no results, because, while "rea" appears four times in the text, it doesn't appear at the end of a word.

You can use `\b` to match the beginning of a word too:

OBSERVE:

```
\bscr
```

You get three matches because you're matching "scr" three times in the three instances of "scream".

Matching Words that End in "es"

Now, you should be able to decipher the term "`[a-z]+es\b`" that we used earlier to match all words ending in "es" from the Sherlock Holmes text.

`[a-z]` says match any letter from a to z. `[a-z]+` says match those letters one or more times so we can find words of any length. Since we're not matching spaces (or other delimiters, like "," or ";"), this locates only single words. But we are finding only words that end in "es" because of the "`es\b`". So `[a-z]+es\b` says, “Find all words with one or more characters matching a-z, ending in es.” (that is, the characters "es" are on a word boundary at the end of the word).

You can do much more with Regular Expression matching. We've hit some of the highlights here, but if you like using Regular Expressions (and you will as you write more code!), you should check out a good Regular Expressions reference book or online page for the various pattern options you can use. There are many of them out there.

Regular Expressions can be hard to read and understand. Keep that in mind as you learn more about Regular Expressions. Because they are so hard to read, using them in your programs can then make your programs more difficult to maintain. Malformed Regular Expressions can cause errors, so use them judiciously, and keep them simple!

Summary of String Properties and Methods

In this lesson you learned lots about JavaScript Strings. Now you know that strings are String objects, with properties and methods, like other JavaScript objects.

Property or Method	What It Does
length	Property: Returns the number of characters in the string.
charAt()	Method: Returns the character at a specific position in the string (remember that strings are 0-based like arrays).
trim()	Method: Removes leading and trailing spaces (but not spaces in the middle).
indexOf()	Method: Returns the position of a substring in the original string, or -1.
toUpperCase()	Method: Returns a new string that is the upper case version of the original string.
toLowerCase()	Method: Returns a new string that is the lower case version of the original string.
split()	Method: Splits a string into substrings based on a split character, returns an array.
match()	Method: Matches a Regular Expression to a string, returns an array.

More about the material in this lesson

The Riverside JS Workshop would like to acknowledge the generosity of O'Reilly Media, Inc. for making this material available to us through the Creative Commons License. We would also like to acknowledge the great work of Elisabeth Robson who authored this content. She is one of our favorites. Elisabeth has written a number of *Head First* programming books for web developers. We recommend them highly. You can browse her work here.