**Excerpt from "Learning to Program" by Steven Foote
(Addison-Wesley Professional, 2014)**

6

Regular Expressions

> **Note**
> **Project:** Validate a phone number using regular expressions.

Most introductory programming books don't include an entire chapter on regular expressions because regular expressions can be intimidating. As a result, many programmers choose to never learn regular expressions. Although other tools can do much of what regular expressions can do, they don't do the job as well. You can drive nails with the backside of a screwdriver, for example, but using a hammer is so much easier. For certain jobs, nothing but regular expressions work. Becoming comfortable with regular expressions early will benefit you as long as you program.

Having said that, I should mention that regular expressions are hard. They are terse and can be quite confusing. They are difficult to document and difficult to read, and making mistakes is easy. Still, they are remarkably useful.

Ctrl+F on Steroids: Looking for Patterns

The last chapter briefly introduced you to regular expressions, but you are probably still wondering what they are and how they are used. Regular expressions are a lot like the Find (keyboard shortcut: Ctrl+F) feature in Word, Chrome, and Sublime Text. Regular expressions search a string for a given set of characters, but they are far more powerful. Imagine that you have a 300-page document and you want to find all the places where a year is mentioned (for example, `2014`). With Find, you have to search for every single year individually—first you'd search for `1900`, then `1901`, then `1902`, and so on before you'd give up as it would take way too long. With regular expressions, you can search for patterns, and a year is a very simple pattern: a sequence of four digits from `0` to `9`. See Listing 6.1.

Listing 6.1 **Searching for Years with Regular Expressions**

```
var yearPattern = /[0-9]{4}/;
```

As you can see, the syntax for regular expressions is a bit strange, but if you look closely, you can see all the relevant pieces. The digits from `0` to `9` are represented by `[0-9]`, and the requirement that there are four of them is represented by `{4}`. So `yearPattern` is really saying, "Any number from `0` to `9`, repeated four times." A perfect regular expression will match everything you want to find in a string and not match anything you don't want to find. For example, `yearPattern` will match any four-digit sequence of numbers, so it will match all the four-digit years (good), but it will also match phone numbers like `800-555-3456` (bad) because there is a sequence of four digits in phone numbers. In this chapter, you learn the tools you need to turn `yearPattern` into a much better regular expression.

Using Regular Expressions in JavaScript

Regular expressions do nothing without a string to test them on. Let's start with a simple string and a simple regular expression. Open the Chrome Dev Tools on any page, and create a string with your name in it; then create a regular expression with your name in it. See Listing 6.2.

Listing 6.2 **My Name As a Regular Expression**

```
> var myName = 'Steven Foote';
  "Steven Foote"
> var namePattern = /Steven/;
  /Steven/
```

Now you need to test whether the string matches the regular expression. You can do this in a few different ways, each with a slightly different purpose and output.

- `test` returns a Boolean, `true` if the string matches and `false` if it doesn't.
- `exec` either returns an array of the first part of the string that matches or else returns nothing.
- `match` either returns the part (or parts) of the string that match in an array or else returns nothing. `match` is different because it is a method on the string and the regular expression is the argument (see Listing 6.3).

Listing 6.3 **Testing Our Regular Expression/String Combo All Three Ways (in the Console)**

```
> namePattern.test(myName);
  true
> namePattern.exec(myName);
  ["Steven"]
> myName.match(namePattern);
  ["Steven"]
```

Repetition

The regular expression `namePattern` isn't much of a pattern at all; it's no different than using Find. What if I want to match `Steven` or `Steve`? Listing 6.4 shows how inflexible `namePattern` is.

Listing 6.4 **Trying to Match `Steve`**

```
> var myNickname = 'Steve Foote';
  "Steve Foote"
> namePattern.test(myNickname);
  false
```

Dang. My regular expression isn't flexible enough for even my nickname. We can fix that—but be warned, this is where regular expressions start to look weird.

?

Leave behind any preconceptions you might have about what the question mark means. In the land of regular expressions, unassuming characters can leave behind the roles they play in normal life and take on new, meaningful powers. The simple `?` has the power to solve our nickname problem. See Listing 6.5.

Listing 6.5 **The Powerful `?`**

```
> namePattern = /Steven?/;
  /Steven?/
> namePattern.test(myNickname);
  true
> namePattern.text(myName);
  true
```

Magic! The `?` tells the regular expression that the character before the `?` should appear once or not at all. In other words, the new `namePattern` says that the `n` is optional.

+

Right now you're probably thinking "Wooooooooow! Regular expressions are amazing!" Well, maybe you're not quite that excited about it, but guess what? Regular expressions can handle it, no matter how excited you are. In Regular Expression Land, the `+` tells the regular expression to match the preceding character one or more times. See Listing 6.6.

Listing 6.6 **Regular Expressions Can Handle All Levels of Excitement**

```
> var excitementPattern = /Wo+w/;
  /Wo+w/
> var notYetExcited = 'Wow... Regular Expressions are weird :/';
  "Wow... Regular Expressions are weird :/"
> excitementPattern.exec(notYetExcited);
  ["Wow"]
> var startingToGetExcited = 'Wooow. Regular Expressions are... pretty cool.';
  "Wooow. Regular Expressions are... hard."
> excitementPattern.exec(startingToGetExcited);
  ["Wooow"]
> var totallyLovingRegEx = 'Wooooooooooww! Regular Expressions are awesome!!!';
  "Wooooooooooww! Regular Expressions are awesome!!!"
> excitementPattern.exec(totallyLovingRegEx);
  ["Wooooooooow"]
> var excitedButBadAtSpelling = 'Www! Relugar Expresions, I <3 u!';
  "Wwwwww! Relugar Expresions, I <3 u!"
> excitementPattern.exec(excitedButBadAtSpelling);
  null
```

*

The star (sometimes called the Kleene star, after its creator, Stephen Kleene) tells the regular expression to match the preceding character zero or more times. It's truly a regular expression rock star. We can make our `excitementPattern` regular expression flexible enough to even handle bad spelling. See Listing 6.7.

Listing 6.7 **A Regular Expression Rock Star**

```
> excitementPattern = /Wo*w/;
  /Wo*w/
> var excitedButBadAtSpelling = 'Www! Relugar Expresions, I <3 u!';
  "Wwwwww! Relugar Expresions, I <3 u!"
> excitementPattern.test(excitedButBadAtSpelling);
  true
```

Special Characters and Escaping

Now that you've met some of the super-powerful characters of Regular Expression Land, you might want to create a regular expression that matches strings that contain those amazing characters, just to show what a huge fan you are. Give it a shot and check out Listing 6.8.

Listing 6.8 **Looking for Stars, Plus More**

```
> var starPattern = /*/;
  SyntaxError: Unexpected token ILLEGAL
> var plusPattern = /+/;
  SyntaxError: Invalid regular expression: /+/: Nothing to repeat
```

Well, that didn't go well. The first `SyntaxError` is pretty mysterious, but the second error is a bit more helpful. Our `plusPattern` regular expression is invalid because the `+` has no preceding character that should be repeated. But we're trying to match the `+`, not repeat some other character. We need to tell the regular expression that we want to match a literal `+` instead of giving `+` super powers. Do you remember when we ran into this problem in Chapter 1, "Hello World! Writing Your First Program," with the apostrophe inside a string? We needed an escape character then, as we do now. Regular expressions also use the backslash for escaping. See Listing 6.9.

Listing 6.9 **Looking for Stars and Actually Finding Them**

```
> var starPattern = /\*/;
  /\*/
> var aStar = 'Kleene, you\'re a *';
  "Kleene, you're a *."
> starPattern.test(aStar);
  true
```

`{1,10}`: Make Your Own Super Powers

The special characters you just learned about are some of the most commonly used characters in regular expressions, but there's more. If `?`, `+`, and `*` aren't specific enough, you can define your own lengths using curly braces. Remember the `yearPattern` example that matches a sequence of exactly four digits? You can use the curly braces in three ways:

- `{n}` matches the preceding character exactly *n* times.
- `{n,}` matches the preceding character at least *n* times.
- `{m,n}` matches the preceding character at least *m* times, but no more than *n* times.

You can actually use the curly brace syntax to get the same result as `?`, `+`, and `*`. `?` is the same as `{0,1}`, `+` is the same as `{1,}`, and `*` is the same as `{0,}`. We will be using the curly braces extensively in the project in this chapter.

Match Anything, Period

Sometimes you want a pattern that matches anything. For instance, we could make our `excitementPattern` match only strings that contain some form of `Wow` *and* contain an

exclamation point (`!`), but there may be some stuff between the `Wow` and the `!` that we don't really care about. Regular expressions have another super-powerful special character to handle that situation: The period, usually called the dot (as in *dot-com*), matches any character. It's like the regular expression wildcard. You can combine the dot and the star to match zero or more characters, regardless of what those characters are. See Listing 6.10.

Listing 6.10 **Matching Anything**

```
> excitementPattern = /Wo*w.*!/;
  /Wo*w.*!/
> excitementPattern.test(notYetExcited);
  false
> excitementPattern.test(excitedButBadAtSpelling);
  true
```

As our patterns get more complex, it can be useful to visualize what is happening. www.regexper.com/ is an amazing online tool that creates a visualization for any regular expression. Figure 6.1 is the visualization for the `excitementPattern`, in the "railroad" diagram format. Your train starts at the dot on the left, and each of the squares is like a train station. Each time the train stops at a station, exactly one character from the string has to get off the train, and the characters have to get off the train in the correct order. The label on the station describes what kind of character is allowed to get off the train. If the train stops at a station and the next character in line is not allowed to get off, the train derails, which means the string does not fit the pattern defined in the regular expression.

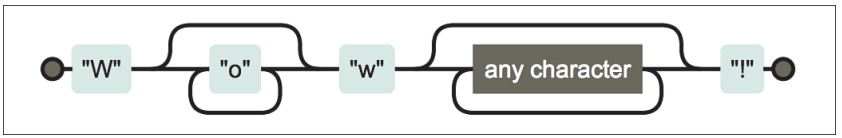

Figure 6.1 A visualization of /Wo*w.*!/

Don't Be Greedy

Regular expressions are greedy—not the *Wall Street*/Gordon Gekko kind of greed, but they are greedy all the same. The regular expression greed means that a regular expression wants to match as much of a string as it can. For example, given the string `'Www! Regular Expressions, I <3 u!'`, the `excitementPattern` (`/Wo*w.*!/`) will match the entire string instead of just the `Www!` part. This is because the `.*!` tells the regular expression to match everything (including the `!`) until it finds the *last* `!`. Sometimes greed is good, but if you want your regular expression not to be greedy, there is hope. Add a `?` (wow, `?` sure has a lot of super powers) after the `*` or `+`, and its greed will go away.

Understanding Brackets from `[A-Za-z]`

The powerful `.` is great when you want to match anything and everything, but sometimes "anything" is too broad. Using brackets, you can define exactly which characters you want to match. If I want to match my name whether or not the `S` is capitalized, I can use brackets to match either `S` or `s`. See Listing 6.11.

Listing 6.11 **Use Brackets to Handle Lowercase Names**

```
> lowerCaseName = 'steven foote';
  "steven foote"
> namePattern     // remember what the name pattern looks like
  /Steven?/
> namePattern.test(lowerCaseName);
  false
> namePattern = /[Ss]teven?/;
  /[Ss]teven?/
> namePattern.test(lowerCaseName);
  true
```

Lists of Characters

The simplest way to use brackets is to list all the acceptable characters within the brackets. In the updated `namePattern`, the acceptable characters are `S` and `s`, so those two characters go inside the brackets. If you wanted to match double or single quotes, you could use `["']`. If you wanted to match any lowercase letter in the English alphabet, you could use `[abcdefghijklmnopqrstuvwxyz]`, and if you wanted to match any lowercase or uppercase letter in the English alphabet, you could use `[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]`. Wow, that is ugly. There must be a better way.

Ranges

There *is* a better way! With the `yearPattern`, we matched any digit from `0` to `9` using `[0-9]`, which is way easier than typing out `[0123456789]`, although they both do the same thing. Within brackets, a dash between two characters is called a range. Ranges are most commonly used for ranges of numbers and letters. Ranges on other characters can be quite confusing, and I don't recommend using them. You can match any upper- or lowercase letter in the English alphabet using `[a-zA-Z]`. That is much better (and you don't have to worry that you missed one of the letters). Ranges don't have to start with `0` or `A`, and they don't have to end with `9` or `Z`. If you want to match only letters in the second half of the alphabet, you can use `[n-z]`, for instance. You can also use a combination of individual characters and ranges: `[12357a-z]`.

A time might come when you want to include a dash as one of the characters in your list, but the dash has super powers within the brackets, so you need to use ... that's right, you need to use an escape character. The pattern `/[a-z\-]/` matches a dash or any lowercase letter. You can also use

brackets and repetition together. The `yearPattern` uses brackets to say that it wants to match digits `0` to `9`; then it uses repetition to say that it wants to match those digits exactly four times.

Negation

If you want to match anything but a group of characters, regular expressions have got you covered with another character with super powers. Add a caret (`^`) at the beginning of the group to match anything but the characters in the group. Let's say that I am adamant about being called Steve instead of Steven. In fact, I would take any character at the end of my name, as long as it's not `n` (I actually prefer to be called Steven, but for the sake of the example, let's pretend). My new `namePattern` could look like Listing 6.12.

Listing 6.12 **Don't Call Me Steven**

```
> namePattern = /Steve[^n]/;
  /Steve[^n]/
> namePattern.test('Steve Foote');
  true
> namePattern.test('Steveq Foote');
  true
> namePattern.test('Steven Foote');
  false
```

A Pattern for Phone Numbers

At this point, you have learned enough to get started on the project of validating phone numbers. First, we need to think of the different valid phone number formats. Here are a few possibilities:

- 1-555-867-5309
- 555-867-5309
- (555)867-5309

We can quickly experiment with these three formats using an online regular expression tool called regexpal: http://regexpal.com/. In the upper box, we enter our regular expression (no need for the `/` when using this tool), and in the lower box, we enter the strings we want to match, one per line. Add each of the three formats to the lower box. You should also add some strings that should not match the pattern, such as `'Regular Expressions are great'` and `'-0-1-2-3-4-5-6'`. These are clearly not phone numbers, so if our pattern matches them, we need to do some work.

In each of these formats, we see some digits and some dashes. In one of these formats, we see parentheses. The formats have between 12 and 14 characters. We can turn that into a rudimentary regular expression using brackets and repetition. Listing 6.13 shows the regular expression, and Figure 6.2 shows a visual representation of the regular expression.

Listing 6.13 **Phone Number Regex, Round 1**

```
var phoneNumberPattern = /[0-9\-\(\)]{12,14}/;
```

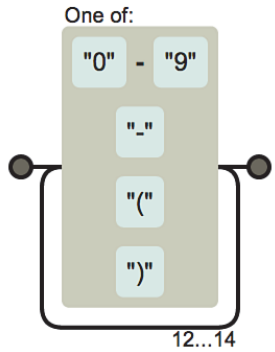

Figure 6.2 A visualization of `phoneNumberPattern`

Our first attempt is great; it matches all three formats. But it also matches one of the non–phone number strings (see Figure 6.3). It's good, but we can do better.

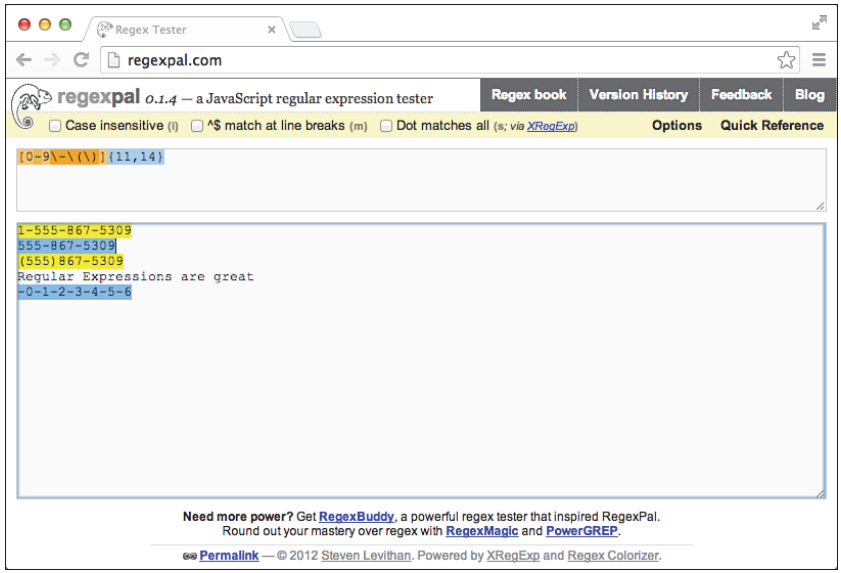

Figure 6.3 A valiant first attempt, but we have a way to go.

To make our pattern even better, we need to break down the phone number into its parts: 1) An optional prefix: `1-`. 2) An optional opening parentheses. 3) A three-digit area code. 4) Either a dash or a closing parentheses. 5) Three digits. 6) A dash. 7) Four digits. Each of these parts is a relatively simple pattern:

1. `1?-?`
2. `\(?`
3. `[0-9]{3}`
4. `[\-\)]`
5. `[0-9]{3}`
6. -
7. [0-9]{4}

Putting them all together (see Listing 6.14), we get a not-so-simple pattern, but we know what each part does, so it's not impossible to understand. With the help of the railroad diagram in Figure 6.4, things get even easier. The real test, though, is whether our new pattern matches everything it should match and rejects everything it should not match (see Figure 6.5).

Listing 6.14 **Phone Number Regex, Round 2**

```
var phoneNumberPattern = /1?-?\(?[0-9]{3}[\-\)][0-9]{3}-[0-9]{4}/;
```

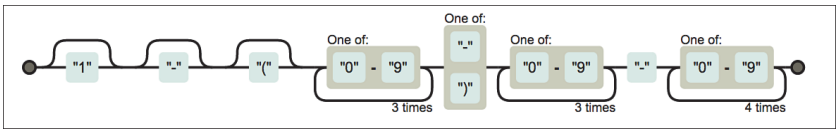

Figure 6.4 A visualization of a better phone number pattern

The phone number pattern is looking great. We will make it even better as we learn more about regular expression features.

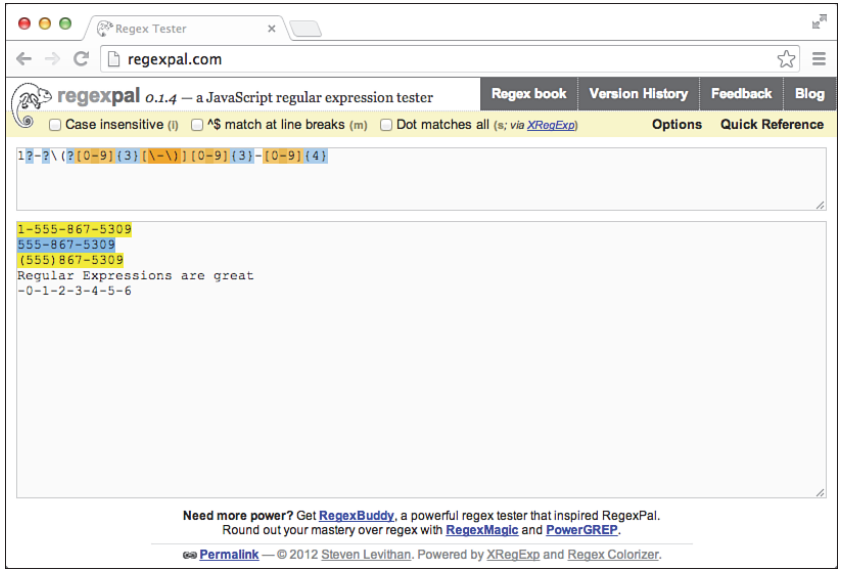

Figure 6.5 A much better phone number pattern

I Need My \s

We have improved our patterns a lot from the first regular expression we wrote (`/Steven/`), and we haven't even gotten into one of the most prolific features regular expressions have to offer: the backslash. So far, we've used the backslash as an escape character, a way to take the super powers away from a special character. But the backslash doesn't just take; the backslash can give super powers to ordinary characters, too.

Shortcuts for Brackets

Most commonly, the backslash turns letters into groups. For instance, `\s` matches any type of whitespace character, such as a space, a tab, and a newline (among others). The backslash can also represent a character that cannot be used literally in a regular expression. For instance, a regular expression generally has to be all on one line, so to represent a new line, you need to use `\n` because you can't just press Enter. The following table describes all the characters that take on special meaning with the backslash.

Symbol	Meaning
`\b`	A word boundary. This matches a place where a word character (see `\w`) is next to a nonword character (see `\W`).
`\B`	A nonword boundary. This matches a place where two word characters are next to each other or two nonword characters are next to each other.

Symbol	Meaning
`\d`	A digit. `\d` is shorthand for `[0-9]`.
`\D`	Anything but a digit. `\D` is shorthand for `[^0-9]`.
`\n`	A newline character.
`\r`	A carriage return character. Used on Windows to create a new line.
`\s`	A whitespace character. Whitespace characters include a space (the character you get when you press the spacebar), a tab, a newline (`\n`), a carriage return (`\r`), and a few others.
`\t`	A tab character.
`\w`	A word character, which is any alphanumeric character or underscore. `\w` is shorthand for `[0-9a-zA-Z_]`.
`\W`	A nonword character, equivalent to `[^0-9a-zA-Z_]`.

With this knowledge, we can make our phone number pattern a little more readable. Our pattern currently repeats `[0-9]` three times. We can replace `[0-9]` with `\d` for a slightly more concise regular expression without changing what it does (see Listing 6.15).

Listing 6.15 **Phone Number Pattern, Round 3**

```
var phoneNumberPattern = /1?-?\(?\d{3}[\-\)]\d{3}-\d{4}/;
```

Let's add some code to our kittenbook project. In `prompt.js`, we'll ask users for their phone numbers instead of their names. Then we'll validate the phone numbers using our regular expression (see Listing 6.16). As you learned in Chapter 2, "How Software Works," we could send an SMS to that number to further our validation. However, that is beyond the scope of this book. Let's just stick to validation via regular expressions for now.

Listing 6.16 **`prompt.js` Now Asks for and Validates Phone Numbers**

```
// Get the user's name.
var userName = prompt('Hello, what\'s your name?');
// Get the user's phone number.
var phoneNumber = prompt('Hello ' + userName + ', what\'s your phone number?');
// Create the phone number pattern.
var phoneNumberPattern = /1?-?\(?\d{3}[\-\)]\d{3}-\d{4}/;
// Create a variable to store the output.
var output = '<h1>Hello, ' + userName + '!</h1>';

// Is the phone number valid?
if (phoneNumberPattern.test(phoneNumber)) {

  // Yes, the phone number is valid! Add the success message to the output.
```

```
  output = output + '<p>' + kbValues.projectName + ' ' + kbValues.versionNumber +
           ' viewed on: ' + kbValues.currentTime + '</p>';

} else {
  // No, the phone number is not valid. Tell the user about the problem.
  output = output + '<h2>That phone number is invalid: ' + phoneNumber;
}
// Insert the output into the web page.
document.body.innerHTML = output;
```

Limitations

Some of these normal characters endowed with super powers are absolutely necessary because they represent a single character (such as `\n` and `\t`) that cannot be represented in any other way in a regular expression. However, some of these characters are just shortcuts to allow you to write less code (such as `\w` and `\s`). The shortcuts are useful, as long as you recognize their limitations. The `\w` matches only the letters from the English alphabet. Throw in just a little bit of Spanish flair to your string (as with `olé`), and your regular expression will fail. Also, `\s` matches a lot of space characters that you might not actually want to match. In both cases, writing out the exact set of characters you want to match (within brackets) is sometimes necessary. The shortcuts are useful—until they're not.

(?:Groups)

I will admit that, up to this point, regular expressions might seem a little boring. Although I do think it's pretty cool to be able to create a pattern that can match any phone number in a whole bunch of different formats, groups (and especially capturing groups) is what made me really love regular expressions. Groups are like subpatterns within the larger regular expression, and the super characters can operate on the entire group together. A few examples should help make this clearer. Let's you have a dog named fifi, and you want to create a regular expression that matches any string with your dog's name in it. To complicate matters, let's say you sometimes call your dog "fi" for short. You need a group, which is a set of characters surrounded by parentheses. A noncapturing group, which is the type we want to use to find `fifi`, starts with `?:`. See Listing 6.17.

Listing 6.17 **Finding** `fifi`

```
> var fifiPattern = /(?:fi){1,2}/;
  /(?:fi){1,2}/
> var fifi = 'I have a dog named fifi';
  "I have a dog named fifi"
> var fi = 'Sometimes I call my dog fi';
  fi = "Sometimes I call my dog fi"
> fifiPattern.test(fifi);
  true
```

```
> fifiPattern.test(fi);
  true
```

What about the `namePattern` to match `'Steven'` and `'Steve'`—what if we want to match `'Stephen'`, too? We can make it happen with a group, but we'll also need to introduce another super character: `|`. The `|`, or pipe (found on the Backslash key, between the Backspace and Enter keys), means "or" in regular expressions. The pipe splits the regular expression and matches either what is on the left or what is on the right. For example, `/apple|broccoli/` matches `'apple juice'` and `'baked broccoli'`, but not `'carrot cake'`. We can use a pipe inside a group to match either `v` for `'Steven'` or `ph` for `'Stephen'`. See Listing 6.18.

Listing 6.18 **Searching for `'Stephen'`**

```
> var namePattern = /Ste(?:v|ph)en?/;
  /Ste(?:v|ph)en?/
> var myName = 'Stephen';
  "Stephen"
> namePattern.test(myName);
  true
> myName = 'Steven';
  "Steven"
> namePattern.test(myName);
  true
```

As a final example, let's consider the prefix in our `phoneNumberPattern`. It's almost right, but it has a slight problem. When we divided the phone number into parts, the first part was an optional prefix of `1-`, which we achieved with `1?-?`. The problem is that `1?-?` matches phone numbers that look like `-877-555-1234` because the pattern matches the `1` zero or one times, and then matches the `-` zero or one times. We really want to match them together zero or one times, and we can do that with a group. See Listing 6.19.

Listing 6.19 **Fixing the Prefix**

```
var phoneNumberPattern = /(?:1-)?\(?\d{3}[\-\)]\d{3}-\d{4}/;
```

That's much better. Another point to note in the `phoneNumberPattern` is the backslashes in front of the nongroup parentheses. Now that you know that parentheses are special characters, those backslashes should make a little more sense.

(Capture)

Capturing groups are (for me, anyway) the most interesting and useful feature of regular expressions. They allow you to extract data from your matched string. For instance, we can use capturing on the `phoneNumberPattern` to extract just the area code from the string. Capturing groups are surrounded by just parentheses (with no special characters such as `?:` at

the beginning). You need to use `match` or `exec` to be able to access what the group captured. See Listing 6.20.

Listing 6.20 **Capture the Area Code**

```
var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
```

Now that we have access to the area code, we can create a more personalized message for our users. We'll assume that the user lives in the location related to the area code (cellphones have kind of ruined this assumption, but we'll go with it anyway). You can add an `areaCodes` object to the `kbValues` object in `values.js`. See Listing 6.21.

Listing 6.21 **Add an `areaCodes` Object to `kbValues`**

```
var kbValues = {
  projectName: 'kittenbook',
  versionNumber: '0.0.1',
  areaCodes: {
    '408': 'Silicon Valley',
    '702': 'Las Vegas',
    '801': 'Northern Utah',
    '765': 'West Lafayette',
    '901': 'Memphis',
    '507': 'Rochester, MN'
  }
};
```

You can add as many area codes as you want to the `areaCodes` object. Now you can update `prompt.js` to use the area code from the user's phone number to create a more personalized greeting. The first step is to update the regular expression to include the capturing group for the area code (see Listing 6.20). Then you need to capture the matches using `exec` or `match`, which will return an array with the area code in index position 1. See Listing 6.22.

Listing 6.22 **Capture the Area Code**

```
// Create the phone number pattern.
var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
// Get matches from phoneNumber
var phoneMatches = phoneNumberPattern.exec(phoneNumber);
// If the phone number is 901-555-5309, then phoneMatches will be
// ['901-555-5309', '901']
var areaCode = phoneMatches[1];
```

Finally, you need to add the personalized message to the output. This brings us to an interesting problem. You know how to access the different attributes of an object using the object

name, then a dot, and then the attribute name, as in `kbValues.projectName`. So we can access the area codes object with `kbValues.areaCodes`, and we have the area code we want stored in the `areaCode` variable. But how can we access the value related to that area code? We could try `kbValues.areaCodes.areaCode`, but that would look for an attribute whose name is `areaCode` inside the `areaCodes` object. In JavaScript, you can use the bracket syntax to access attributes of an object (see Listing 6.23). It looks similar to the syntax for accessing items in an array. Listing 6.24 shows how to use the bracket syntax to get the location based on area code.

Listing 6.23 **Accessing Object Attributes with the Bracket Syntax**

```
> var states = {'AL': 'Alabama',
                'AK': 'Alaska',
                'AZ': 'Arizona',
                'AK': 'Arkansas'};
> states.AL;
  'Alabama'
> var stateName = 'AL';
  'AL'
> states.stateName;  // This won't work :(
  undefined
> states[stateName];
  'Alabama'
> states['AL'];
  'Alabama'
```

Listing 6.24 **Getting the Location Based on the `areaCode`**

```
// Get the location using bracket syntax
var userLocation = kbValues.areaCodes[areaCode];
```

Now you can add the `location` to the output in whatever way you want. Maybe you could ask how the weather in `location` is these days. Regardless, your users are going to be impressed that you figured out where they live based on their phone number. They would be even more impressed if they knew all the regular expression magic you had to do to figure out their location.

Capture the Tag

Now that you've seen a bit about how using regular expressions and capturing groups work, it's time to try some out on your own. Open the Chrome Dev Tools on any page (I recommend the Mozilla Developer Network, but any page will do). Find an HTML tag in the Elements tab, copy the HTML of that element (see Figure 6.6), and save the string to a variable in the console (I recommend an element that has no children so that you don't have to deal with newlines when saving the string to a variable). Now write a regular expression that will capture the tag

name (for example, the tag name of `<h2>` is `h2`). You should be able to use the same regular expression to capture the tag name of any HTML tag you copy.

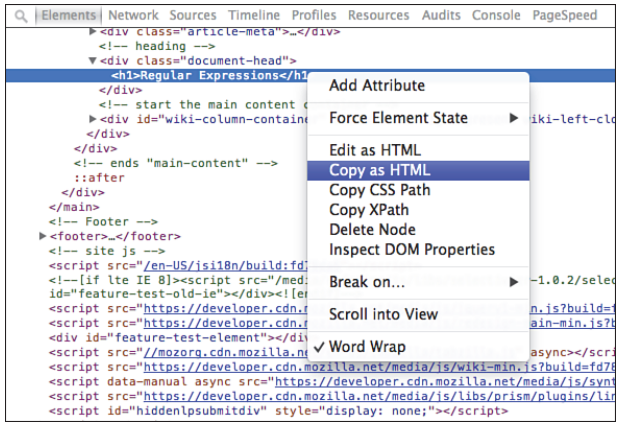

Figure 6.6 Copy the HTML of a given tag.

Advanced Find and Replace

To see another way capturing groups can be useful, let's think back to Listing 5.9 of the last chapter. That was the one where we updated the age attribute of the `author` string with the long, complex code. We can improve that long, complex code a lot by using a regular expression with a capturing group and also using advanced find and replace (see Listing 6.25).

Listing 6.25 **Improving the String Data Structure with a Capturing Group**

```
var author = 'firstName=Steven&lastName=Foote&age=27&favoriteFoods=waffles,Thai
curry';

// Use the String's built-in replace method.
author.replace(/(age=)(\d+)/, function(fullMatch, group1, group2) {
  /**
    * fullMatch contains "age=27"
    * group1 contains "age="
    * group2 contains "27"
    */

  // Whatever we return will replace the entire match in the original string
  // Add 1 to group2, then put the two groups back together.
  // parseInt turns the string "27" into the number 27, so we can add them together.
  // Without parseInt, "27" + 1 becomes "271", not 28.
  return group1 + (parseInt(group2, 10) + 1);
```

```
});
// author is now 'firstName=Steven&lastName=Foote&age=28&favoriteFoods=waffles,Thai
curry'
```

The Beginning and the End (of a Line)

Find and replace with regular expressions is pretty amazing, but one part of the previous regular expression is not quite right. The regular expression matches `'age='` and then some numbers. What if the author string also contained `'page=23'`? Then our regular expression would match `'age=27'` and `'page=23'`. We could change our regular expression to be `/(&age=)(\d+)/`, but we would still have a problem: What if `'age'` is at the beginning of the string and there is no preceding `&`? Fortunately, regular expressions have a way of identifying the beginning (and end) of a line. The beginning of a line is represented by a `^` (remember, the `^` has a different meaning inside brackets—regular expressions sure can be confusing), and the end of the line is represented by a `$`. We can change our find-and-replace regular expression to match either `&` or the beginning of the line, then `'age='`: `/((?:^|&)age=)(\d+)/`.

Flags

Regular expressions have a few default behaviors that you can turn on or off using flags. These default behaviors usually work just fine, but when they don't, it's nice to be able to turn them off.

Global

By default, as soon as a regular expression finds a match within a string, it stops looking. The global flag tells the regular expression to find all matches in the string, not just the first. To set the global flag, you place a `g` outside the closing forward slash of your regular expression: `/[0-9]{4}/g` finds all years in a string, not just the first one.

Ignore Case

Regular expressions are case sensitive by default, but sometimes you don't care whether text is upper or lower case. You can keep your regular expressions from being case sensitive using the ignore case flag (set with an `i`, so `/ste(?:ph|v)en/gi` finds every instance of `'Steven'`, `'steven'`, `'Stephen'`, and `'stephen'` in a string). Note how the global flag and the ignore case flag can be (but don't have to be) used together.

Multiline

By default, `^` and `$` match only the beginning of the string and the end of the string, respectively. In other words, if your string contains multiple lines, the beginning and ends of the lines are not actually matched by `^` and `$` unless you turn on the multiline flag. The multiline flag is set with an `m`. `/.*\.html$/gm` matches every line in a string that ends with `'.html'`.

When Will You Ever Use Regex?

By now you are probably thinking, "I never want to see another regular expression again. My head hurts." Or maybe you're thinking, "Wow! I <3 regular expressions!" I'm guessing it's not the latter. Either way, you've learned a lot about regular expressions, and you might be wondering about when and how you will actually use them. The following examples are a few of the ways that I have actually used regular expressions to make my work easier (for real, regular expressions can actually make life easier, not harder).

`grep`

In Chapter 3, "Getting to Know Your Computer," you learned about the command-line tool `grep`. At the time, I made `grep` seem like Find for searching multiple files at once. But `grep` is actually all about regular expressions (in fact, *grep* stands for global regular expression print). Using `grep`, you can search for patterns within multiple files at once. When working on a project that has a lot of files (LinkedIn has a *lot* of files), `grep` is a necessity. For example, I can search all my CSS files for places where I am using `float: left;` or `float: right;` using one `grep` command.

Code Refactoring

I am writing this book using HTML, and I have used regular expressions (and especially advanced find and replace) to help me work faster. As I start a new chapter, I copy the portion of my outline that relates to that chapter into a new HTML file. The items in the outline use HTML lists (`<ul>` and `<li>` tags), but these items become headers in the actual chapter (`<h1>`, `<h2>`, and so on). I could switch each of these tags by hand, but instead I write a couple regular expressions. As shown in Figure 6.7, Sublime Text, Vim, and many other editors support regular expressions (and capturing groups) in their find-and-replace features.

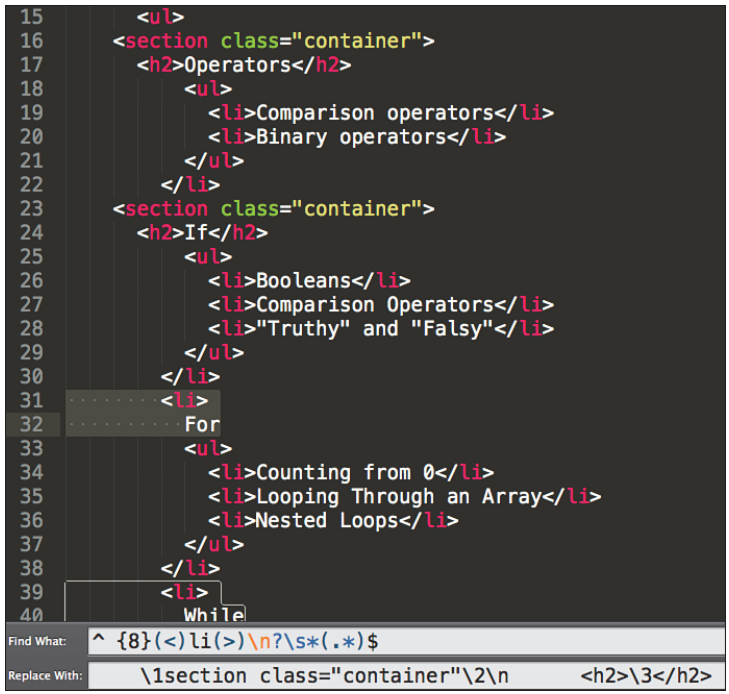

Figure 6.7 Change list items to headers with regular expressions

Validation

In this chapter, you have written a pretty good phone number validation (for U.S. phone numbers). Nearly every time I have to do validation, I use regular expressions. Emails are especially difficult to validate; do a search for "email regular expression," and you will see what I mean. Again, just because a phone number or email address matches a regular expression does not mean that it is valid, but regular expressions can tell you whether the string at least *looks* reasonable.

Data Extraction

My very first programming project was to use regular expressions to extract data about books from a very large (1MB) text file. That project could be the reason I am particularly fond of capturing groups. When I saw regular expressions turn an unusable 30,000-page report into a useful spreadsheet, I was hooked.

Summing Up

You have learned a lot in this chapter. What once looked like random characters (ahem, `/(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/`, ahem) now makes sense to you. Sure, it might take some time to figure out what it means (and I think it always will), but you can figure it out. Regular expressions are a great tool, but remember that just because you're holding a hammer doesn't make everything a nail. In this chapter, you learned about:

- The regular expression syntax
- Quantifiers such as `?`, `+`, and `*`
- Character sets in brackets
- Groups and capturing groups
- Advanced find and replace and other applications of regular expressions

In the next chapter, you will learn about:

- Operators
- Programming with flow control
- How to use `if`, `for`, `while`, and `switch` to control flow
- Another way to code defensively with `try`
- Triggers and events