9
Prompts

A *prompt box* asks the user for some information and provides a response field for her answer.

This code asks the user the question "Your species?" and provides a default answer in the response field, "human". She can change the response. Whether she leaves the default response as-is or changes it to something else, her response is assigned to the variable.

```
var spec = prompt("Your species?", "human");
```

Prompt code is like alert code, with two differences.

In a prompt, you need a way to capture the user's response. That means you need to start by declaring a variable, followed by an equal sign.

In a prompt, you can specify a second string. This is the default response that appears in the field when the prompt displays. If the user leaves the default response as-is and just clicks **OK**, the default response is assigned to the variable. It's up to you whether you include a default response.

As you might expect, you can assign the strings to variables, then specify the variables instead of strings inside the parentheses.

```
var question = "Your species?";
var defaultAnswer = "human";
var spec = prompt(question, defaultAnswer);
```

The user's response is a text string. Even if the response is a number, it comes back as a string. For example, consider this code.

```
var numberOfCats = prompt("How many cats?");
var tooManyCats = numberOfCats + 1;
```

Since you're asking for a number, and the user is presumably entering one, you might expect the math in the second statement to work. For example, if the user enters 3, the variable **`tooManyCats`** should have a value of 4, you might think. But this is not the result we’ll get. All responses to prompts come back as strings. When the string, "3", is linked with a plus to the number, 1, JavaScript converts the 1 to a string and concatenates. So the value of **`tooManyCats`** winds up being not 4 but "31". You'll learn how to solve this problem in a subsequent chapter.

If the user enters nothing and clicks OK, the variable is assigned an empty string: ""

If the user clicks **Cancel**, the variable is assigned a special value, null.

Coding Alternatives to Be Aware Of

- Some coders write **`window.prompt`** instead of, simply, **`prompt`**. This is a highly formal but perfectly correct way to write it. Most coders prefer the short form. We'll stick to the short form in this training.
- In the example above, some coders would use single rather than double quotation marks. This is legal, as long as it's a matching pair. But in a case like this, I'll ask you to use double quotation marks.

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/9.html

10
if statements

Suppose you code a prompt that asks, "What does a dog wag?"
If the user answers correctly, you display an alert congratulating him.
This is the code.

```
var x = prompt("What does a dog wag?");
if (x === "tail") {
  alert("Correct!");
}
```

If the user enters "tail" in the prompt field, the congratulations alert displays. If he enters something else, nothing happens. (This simplified code doesn't allow for other correct answers, like "his tail." I don't want to get into that now.)

There's a lot to take in here. Let's break it down.

An if statement always begins with **`if`**. The space that separates it from the parenthesis is new to you. I've taught you to code alerts and prompts with the opening parenthesis running up against the keyword: **`alert("Hi");`** Now I'm asking you *not* to do that in an *if* statement. It's purely a matter of style, but common style rules sanction this inconsistency.

Following the **`if`** keyword-plus-space is the condition that's being tested—does the variable that's been assigned the user's response have a value of "tail"?

The condition is enclosed in parentheses.

If the condition tests true, something happens. Any number of statements might execute. In this case, only one statement executes: a congratulatory alert displays.

Following my style rules, the statement or statements that execute if the condition tests true are indented 2 spaces.

The first line of an *if* statement ends with an opening curly bracket. An entire *if* statement ends with a closing curly bracket on its own line. Note that this is an exception to the rule that a statement ends with a semicolon. It's common to omit the semicolon when it's a complex statement that's paragraph-like and ends in a curly bracket.

But what about that triple equal sign? You might think that it should just be an equal sign, but the equal sign is reserved for *assigning* a value to a variable. If you're *testing* a variable for a value, you can't use the single equal sign.

If you forget this rule and use a single equal sign when you should use the triple equal sign, the code won't run properly.

As you might expect, you can use a variable instead of a string in the example code.

```
var correctAnswer = "tail";
if (x === correctAnswer) {
  alert("Correct!");
}
```

When a condition is met, you can have any number of statements execute.

```
var correctAnswer = "tail";
if (x === correctAnswer) {
  score++;
  userIQ = "genius";
  alert("Correct!");
}
```

Coding Alternatives to Be Aware Of

- Some coders write simple if statements without curly brackets, which is legal. Some put the opening curly bracket on its own line. Some put the whole if statement, if it's simple, on a single line. I find it easiest not to have to make case-by-case decisions, so I format all if statements the same way, as shown in the example. In the exercises, I'll ask you to follow these style rules for all *if* statements.
- In most cases, a double equal sign **==** is just as good as a triple equal sign **===**. However, there is a slight technical difference, which you may never need to know. Again, to keep things simple, I always use the triple equal sign.

Find the interactive coding exercises for this chapter at:
http://www.ASmarterWayToLearn.com/js/10.html

11
Comparison operators

Let's talk a little more about **===**. It's a type of *comparison operator*, specifically an *equality operator*. As you learned in the last chapter, you use it to compare two things to see if they're equal.

You can use the equality operator to compare a variable with a string, a variable with a number, a variable with a math expression, or a variable with a variable. And you can use it to compare various combinations. All of the following are legal first lines in *if* statements:

```
if (fullName === "Mark" + " " + "Myers") {
if (fullName === firstName + " " + "Myers") {
if (fullName === firstName + " " + lastName) {
if (totalCost === 81.50 + 135) {
if (totalCost === materialsCost + 135) {
if (totalCost === materialsCost + laborCost) {
if (x + y === a - b) {
```

When you're comparing strings, the equality operator is case-sensitive. "Rose" does not equal "rose."

Another comparison operator, **!==**, is the opposite of **===**. It means is not equal to.

```
if (yourTicketNumber !== 487208) {
  alert("Better luck next time.");
}
```

Like **===**, the not-equal operator can be used to compare numbers, strings, variables, math expressions, and combinations.

Like **===**, string comparisons using the not-equal operator are case-sensitive. It's true that "Rose" !== "rose".

Here are 4 more comparison operators, usually used to compare numbers.

> is greater than

< is less than

>= is greater than or equal to

<= is less than or equal to

In the examples below, all the conditions are true.

```
if (1 > 0) {
if (0 < 1) {
if (1 >= 0) {
if (1 >= 1) {
if (0 <= 1) {
if (1 <= 1) {
```

Coding Alternatives to Be Aware Of

Just as the double equal sign can usually be used instead of the triple equal sign, **`!=`** can usually be used instead of **`!==`**. In the exercises, I'll ask you to stick to **`!==`**.

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/11.html

12
if...else and *else if* statements

The *if* statements you've coded so far have been all-or-nothing. If the condition tested true, something happened. If the condition tested false, nothing happened.

```
var x = prompt("What does a dog wag?");
if (x === "tail") {
  alert("Correct!");
}
```

Quite often, you want something to happen either way. For example:

```
var x = prompt("What does a dog wag?");
if (x === "tail") {
  alert("Correct!");
}
if (x !== "tail") {
  alert("Wrong answer");
}
```

In this example, we have two *if* statements, one testing for "tail," and another testing for not-"tail". So all cases are covered, with one alert or another displaying, depending on what the user has entered.

The code works, but it's more verbose than necessary. The following code is more concise and, as a bonus, more readable.

```
if (x === "tail") {
  alert("Correct!");
}
else {
  alert("Wrong answer");
}
```

In the style convention I follow, the **`else`** part has exactly the same formatting as the **`if`** part.

As in the **`if`** part, any number of statements can execute within the **`else`** part.

```
var correctAnswer = "tail";
if (x === correctAnswer) {
  alert("Correct!");
}
else {
  score--;
  askAgain = "yes";
  alert("Incorrect");
}
```

`else if` is used if all tests above have failed and you want to test another condition.

```
var correctAnswer = "tail";
if (x === correctAnswer) {
  alert("Correct!");
}
else if (x === "tale") {
  alert("Incorrect but close");
}
else {
  alert("Incorrect");
}
```

In a series of if tests, JavaScript stops testing whenever a condition tests true.

Coding Alternatives to Be Aware Of

There are so many ways to format if statements and their variations that the range of possibilities is almost endless. I'm partial to the format I've showed you, because it's easy to learn and produces readable code. I'll ask you to stick to this format throughout the exercises.

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/12.html

13
Testing sets of conditions

Using the *if* statement, you've learned to test for a condition. If the condition is met, one or more statements execute. But suppose not one but two conditions have to be met in order for a test to succeed.

For example, if a guy weighs more than 300 pounds, he's just a great big guy. But if he weighs more than 300 pounds and runs 40 yards in under 6 seconds? You're going to invite him to try out for the NFL as a lineman. You can test for a combination of conditions in JavaScript by using...

&& —which is the equivalent of *and* in English. Here's the code.

```
if (weight > 300 && time < 6) {
  alert("Come to our tryout!");
}
else {
  alert("Come to our cookout!");
}
```

You can chain any number of conditions together.

```
if (weight > 300 && time < 6 && age > 17 && gender === "male") {
  alert("Come to our tryout!");
}
else {
  alert("Come to our cookout!");
}
```

You can also test for any of a set of conditions The operator is **||** — which is the equivalent of *or* in English. Here's an example.

```
1 if (SAT > avg || GPA > 2.5 || sport === "football") {
2   alert("Welcome to Bubba State!");
3 }
4 else {
5   alert("Have you looked into appliance repair?");
6 }
```

If in line 1 any or all of the conditions are true, the first alert displays. If none of them are true (line 4), the second alert displays.

You can combine any number of **and** and **or** conditions. When you do, you create ambiguities. Take this line...

```
if (age > 65 || age < 21 && res === "U.S.") {
```

This can be read in either of two ways.

The first way it can be read: If the person is over 65 or under 21 and, in addition to either of these conditions, is also a resident of the U.S. Under this interpretation, both columns need to be true in the following table in order for the overall *if* statement to be true...

Over 65 or under 21	Resident of U.S.

The second way it can be read: If the person is over 65 and living anywhere or is under 21 and a resident of the U.S. Under this interpretation, if either column in the following table is true, the overall *if* statement is true.

Over 65	Under 21 and U.S. resident

It's the same problem you face when you combine mathematical expressions. And you solve it the same way: with parentheses.

In the following code, if the subject is over 65 and a U.S. resident, it's a pass. Or, if the subject is under 21 and a U.S. resident, the overall *if* statement is true.

```
if ((age > 65 || age < 21) && res === "U.S.") {
```

In the following code, if the subject is over 65 and living anywhere, it's a pass. Or, if the subject is under 21 and living in the U.S., the overall *if* statement is true.

```
if (age > 65 || (age < 21 && res === "U.S.")) {
```

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/13.html

14
if statements nested

Check out this code.

```
if ((x === y || a === b) && c === d) {
 g = h;
}
else {
  e = f;
}
```

In the code above, if either of the first conditions is true, and, in addition, the third condition is true, then **`g`** is assigned **`h`**. Otherwise, **`e`** is assigned **`f`**.

There's another way to code this, using nesting.

```
1  if (c === d) {
2    if (x === y) {
3       g = h;
4    }
5    else if (a === b) {
6      g = h;
7    }
8    else {
9      e = f;
10   }
11 }
12 else {
13   e = f;
14 }
```

Nest levels are communicated to JavaScript by the positions of the curly brackets. There are three blocks nested inside the top-level **`if`**. If the condition tested by the top-level **`if`**—that **`c`** has the same value as **`d`**—is false, none of the blocks nested inside executes. The opening curly bracket on line 1 and the closing curly bracket on line 11 enclose all the nested code, telling JavaScript that everything inside is second-level.

For readability, a lower level is indented 2 spaces beyond the level above it.

In the relatively simple set of tests and outcomes shown in this example, I would prefer to use the more concise structure of multiple conditions. But when things get really complicated, nested ifs are a good way to go.

Find the interactive coding exercises for this chapter at
http://www.ASmarterWayToLearn.com/js/14.html

15
Arrays

Let's assign some string values to some variables.

```
var city0 = "Atlanta";
var city1 = "Baltimore";
var city2 = "Chicago";
var city3 = "Denver";
var city4 = "Los Angeles";
var city5 = "Seattle";
```

The variable names are all the same, except they end in different numbers. I could have named the variables **`buffy`**, **`the`**, **`vampireSlayer`**, and so on if I'd wanted to, but I chose to name them this way because of where this discussion is going.

Now, having made these assignments, if I code...

```
alert("Welcome to " + city3);
```

...an alert displays saying, "Welcome to Denver". I'm going to show you another type of variable, one that will come in handy for many tasks that you'll learn about in later chapters. I'm talking about a type of variable called an *array*. Whereas an ordinary variable has a single value assigned to it—for example, 9 or "Paris"—an array is a variable that can have multiple values assigned to it. You define an array this way:

```
var cities = ["Atlanta", "Baltimore", "Chicago", "Denver",
"Los Angeles", "Seattle"];
```

In the example at the beginning of this chapter, I ended each variable name with a number. **`city0`** was "Atlanta", **`city1`** was "Baltimore", and so on. The array I just defined is similar, but in the case of an array defined the way I just defined one, JavaScript numbers the different values, or elements, automatically. (You can control the numbering yourself by defining elements individually. See below.) And you refer to each element by writing the array name—cities in this case—followed by a number enclosed in square brackets. **`cities[0]`** is "Atlanta", **`cities[1]`** is "Baltimore", and so on.

Because JavaScript automatically numbers array elements, you have no say in the numbering. The first element in the list always has an *index* of 0, the second element an index of 1, and so on.

This is the alert I coded above, but now specifying an array element instead of an ordinary variable.

```
alert("Welcome to " + cities[3]);
```

An array can be assigned any type of value that you can assign to ordinary variables. You can even mix the different types in the same array (not that you would ordinarily want to).

```
var mixedArray = [1, "Bob", "Now is", true];
```

In the example above, **`mixedArray[0]`** has a numerical value of 1, **`mixedArray[1]`** has a string value of "Bob", and so on.

Things to keep in mind:

- The first item always has an index of 0, not 1. This means that if the last item in the list has an index of 9, there are 10 items in the list.
- The same naming rules you learned for ordinary variables apply. Only letters, numbers, $ and _ are legal. The first character can't be a number. No spaces.
- Coders often prefer to make array names plural—cities instead of city, for example—since an array is a list of things.
- Like an ordinary variable, you declare an array only once. If you assign new values to an array that has already been declared, you drop the var.

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/15.html

16
Arrays: Adding and removing elements

As you learned in earlier chapters, you can declare an empty variable, one that doesn't have a value. Then you can assign it a value whenever you like. And you can change its value at will. You can do all these things with an array, as well.

This is how you declare an empty array.

```
var pets = [];
```

Assume that the array `pets` has already been declared. This is how you assign values to it.

```
pets[0] = "dog";
pets[1] = "cat";
pets[2] = "bird";
```

In the example above, I defined the first three elements of the array, in order. But you can legally leave gaps in an array if you choose to (not that you normally would). For example, suppose you start with the same empty array and code these lines.

```
pets[3] = "lizard";
pets[6] = "snake";
```

Now, if you refer to `pets[3]`, you'll get "lizard". If you refer to `pets[6]`, you'll get "snake". But if you refer to pets`[0]` through `pets[2]` or `pets[4]` or `pets[5]`, you'll get `undefined`.

You can assign additional values to an array that already has values. Assume that the first three elements of the array pets are "dog", "cat", and "bird". Then you write this code.

```
pets[3] = "lizard";
pets[4] = "fish";
pets[5] = "gerbil";
pets[6] = "snake";
```

Now the array has 7 elements: "dog", "cat", "bird", "lizard", "fish", "gerbil", and "snake".

If you assign a new value to an array element that already has one, the old value is replaced by the new one.

Using the keyword, **`pop`**, you can remove the last element of an array.

Suppose you have an array, **`pets`**, whose elements are "dog", "cat", and "bird". The following code deletes the last element, "bird", leaving a two-element array.

```
pets.pop();
```

You can assign the deleted last element to a variable.

```
var lastElement = pets.pop();
```

Now the variable **`lastElement`** represents the string “bird.”

Using the keyword, **`push`**, you can add one or more elements to the end of an array.

Suppose you have that same array consisting of "dog", "cat", and "bird". The following code adds two new elements to the end of the array.

```
pets.push("fish", "ferret");
```

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/16.html

17
Arrays: Removing and inserting elements

Use the **shift** method to remove an element from the beginning of an array.

Suppose you have an array, **pets**, whose elements are "dog", "cat", and "bird". The following removes the first element, "dog", leaving you with a two-element array.

```
pets.shift();
```

To add one or more elements to the beginning of an array, use the **unshift** method. The following code adds two elements to the beginning of the array.

```
pets.unshift("fish", "ferret");
```

Use the **splice** method to insert one or more elements anywhere in an array, while optionally removing one or more elements that come after it. Suppose you have an array with the elements "dog", "cat", "fly", "bug", "ox". The following code adds "pig", "duck", and "emu" after "cat" while removing "fly" and "bug".

```
pets.splice(2, 2, "pig", "duck", "emu");
```

The first digit inside the parentheses is the index of the position where you want to start adding if you're adding and deleting if you're deleting. The second digit is the number of existing elements to remove, starting with the first element that comes after the element(s) that you're splicing in. The code above leaves you with an array consisting of "dog", "cat", "pig", "duck", "emu", and "ox".

You could make additions without removing any elements. The following code adds "pig", "duck", and "emu" without removing any elements.

```
pets.splice(2, 0, "pig", "duck", "emu");
```

You can also remove elements without adding any. If you start with the elements "dog", "cat", "fly", "bug", and "ox", the following code removes two elements starting at index 2—"fly" and "bug". This leaves "dog", "cat", and "ox".

```
pets.splice(2, 2);
```

Use the **`slice`** method to copy one or more consecutive elements in any position and put them into a new array. If you start with an array, **`pets`**, consisting of "dog", "cat", "fly", "bug", and "ox", the following code copies "fly" and "bug" to the new array **`noPets`** and leaves the original array, **`pets`**, unchanged.

```
var noPets = pets.slice(2, 4);
```

The first digit inside the parentheses is the index of the first element to be copied. The second digit is the index of the element after the last element to be copied.

Two things could trip you up here:

Since the first index number inside the parentheses specifies the first element to be copied, you might think the second index number specifies the last element to be copied. In fact, the second number specifies the index number of the element *after* the last element to be copied.

You must assign the sliced elements to an array. It could, of course, be the same array from which you're doing the slicing. In that case, you'd be reducing the original array to only the copied elements.

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/17.html

18
for loops

You know the song "99 Bottles of Beer on the Wall"? If you're teaching someone the song, you could give them these instructions:

1. Sing "99 bottles of beer on the wall, 99 bottles of beer."
2. Sing "Take one down and pass it around, 98 bottles of beer on the wall."
3. Sing "98 bottles of beer on the wall, 98 bottles of beer."
4. Sing "Take one down and pass it around, 97 bottles of beer on the wall."
5. Sing "97 bottles of beer on the wall, 97 bottles of beer."
6. Sing "Take one down and pass it around, 96 bottles of beer on the wall."

...and so on, for 192 more lines of instructions.

But that isn't how you'd give the instructions, is it? You'd be more concise. You'd say something like this:

Sing "99 bottles of beer on the wall, 99 bottles of beer. Take one down and pass it around, 98 bottles of beer on the wall." Repeat this, subtracting 1 each time, until there are no more bottles of beer on the wall.

In coding, you run into the bottles-of-beer situation quite often. For example, suppose you've offered to check if the user's city is one of the 5 cleanest in the U.S. The user has entered her city, and you've assigned her city to the variable **`cityToCheck`**.

You've already assigned the list of the 5 cleanest cities to the array **`cleanestCities`**.

```
var cleanestCities = ["Cheyenne", "Santa Fe", "Tucson",
"Great Falls", "Honolulu"];
```

Now you go through the array to see if there's a match with the user's city. If there is, you display an alert telling the user her city is one of the cleanest. If there's no match, you display an alert telling the user her city isn't on the list.

```
if (cityToCheck === cleanestCities[0]) {
  alert("It's one of the cleanest cities");
}
else if (cityToCheck === cleanestCities[1]) {
  alert("It's one of the cleanest cities");
}
else if (cityToCheck === cleanestCities[2]) {
  alert("It's one of the cleanest cities");
}
else if (cityToCheck === cleanestCities[3]) {
  alert("It's one of the cleanest cities");
}
else if (cityToCheck === cleanestCities[4]) {
  alert("It's one of the cleanest cities");
}
else {
  alert("It's not on the list");
}
```

Conveniently, JavaScript provides a more concise coding approach. Here's a *for* loop that accomplishes most of what the verbose code in the example above accomplishes.

```
for (var i = 0; i <= 4; i++) {
  if (cityToCheck === cleanestCities[i]) {
    alert("It's one of the cleanest cities");
  }
}
```

Let me break down the first line for you.

The first line begins with the keyword `for`.

The three specifications that define the loop are inside the parentheses.

1. A variable that counts iterations and also serves as the changing array index is declared and set to a starting value, in this case 0.
2. The limit on the loop is defined. In this case, the loop is to keep running as long as the counter doesn't exceed 4. Since the counter, in this case, is starting at 0, the loop will run 5 times.
3. What happens to the counter at the end of every loop. In this case, the counter is incremented each time.

The three specifications inside the parentheses are always in the same order:

1. What to call the counter (usually **`i`**) and what number to start it at (typically 0)
2. How many loops to run (in this case, the number of elements in the array)
3. How to change the counter after each iteration (typically to add 1 each time through)

Things to keep in mind:

- In the example, the counter, **`i`**, serves two purposes. It keeps track of the number of iterations so looping can halt at the right point. And it serves as the index number of the array, allowing the code to progress through all the elements of the array as the counter increments with each iteration.
- There is nothing sacred about using **`i`** as the counter. You can use any legal variable name. But coders usually use **`i`** because it keeps the first line compact, and because traditionally **`i`** stands for "integer." I think of it as standing for "iteration."
- In the example, the initial count is 0, the index number of the first element of the array. But it could be any number, depending on your needs.
- In the example, the counter increments with each iteration. But, depending on your needs, you can decrement it, increase it by 2 or another number, or change it in some other way each time through.
- In the example, I specify that the loop is to run as long as **`i <= 4`**. Alternatively, I could have specified **`i < 5`**. Either way, since the counter starts at 0, the loop runs 5 times.

Find the interactive coding exercises for this chapter at: http://www.ASmarterWayToLearn.com/js/18.html

19
for loops: Flags, Booleans, array length, and loopus interruptus

There are several problems with the *for* loop example I gave you in the last chapter. The first problem is a potential communication problem. If a match between the user's city and the list of cleanest cities is found, a confirming alert displays. But if there is no match, nothing happens. The user is left in the dark. If no match is found, we need to display an alert saying so. But how do we do that?

We do it with a *flag*. A flag is just a variable that starts out with a default value that you give it, and then is switched to a different value under certain conditions. In our example, let's say we define **`matchFound`** as the flag.

```
var matchFound = "no";
```

If a match is found, the value of the flag is changed. At the end, if the flag hasn't been changed—if it still has the original value of "no"—it means no match was found, and so we display an alert saying the city isn't on the list.

```
var matchFound = "no";
for (var i = 0; i <= 4; i++) {
  if (cityToCheck === cleanestCities[i]) {
    matchFound = "yes";
    alert("It's one of the cleanest cities");
  }
}
if (matchFound === "no") {
  alert("It's not on the list");
}
```

This works, but rather than assigning the strings "no" and "yes" to the switch, it's conventional to use the *Boolean* values **`false`** and **`true`**.

```
var matchFound = false;
for (var i = 0; i <= 4; i++) {
  if (cityToCheck === cleanestCities[i]) {
    matchFound = true;
    alert("It's one of the cleanest cities");
  }
}
if (matchFound === false) {
  alert("It's not on the list");
}
```

There are only two Booleans, **`true`** and **`false`**. Note that they aren't enclosed in quotes.

The next problem with our example is that it potentially wastes computing cycles. Suppose on the second loop a match is found and the alert displays. The way the loop is coded, the loop goes on looping all the way to the end. This is unnecessary, since we got our answer in the second loop. The problem is solved with the keyword **`break`**.

```
var matchFound = false;
for (var i = 0; i <= 4; i++) {
  if (cityToCheck === cleanestCities[i]) {
    matchFound = true;
    alert("It's one of the cleanest cities");
    break;
  }
}
if (matchFound === false) {
  alert("It's not on the list");
}
```

The last problem: In the example, I assume that the number of elements in the array is known. But what if it isn't? JavaScript has a way of finding out. The following code assigns the number of elements in the array **`cleanestCities`** to the variable **`numElements`**.

```
var numElements = cleanestCities.length;
```

Now we can limit the number of loops to the count that JavaScript comes up with.

```
var numElements = cleanestCities.length;
var matchFound = false;
for (var i = 0; i < numElements; i++) {
  if (cityToCheck === cleanestCities[i]) {
  matchFound = true;
  alert("It's one of the cleanest cities");
  break;
  }
}
if (matchFound === false) {
  alert("It's not on the list");
}
```

Now the loop keeps going as long as **`i`** is less than the number of elements. (Since the length number is 1-based and the **`i`** number is 0-based, we need to stop 1 short of the length.)

Find the interactive coding exercises for this chapter at:
http://www.ASmarterWayToLearn.com/js/19.html

20
for loops nested

Atlantic Records has hired you and me to generate a list of names for future rap stars. To make things easy, we'll start by making separate lists of some first names and last names.

First Names	Last Names
BlueRay	Zzz
Upchuck	Burp
Lojack	Dogbone
Gizmo	Droop
Do-Rag	

By combining each of the first names with each of the last names, we can generate 20 different full names for rappers.

Starting with "BlueRay," we go through the list of last names, generating...

BlueRay Zzz
BlueRay Burp
BlueRay Dogbone
BlueRay Droop

We move to the next first name, "Upchuck." Again, we go through the list of last names, generating...

Upchuck Zzz
Upchuck Burp
Upchuck Dogbone
Upchuck Droop

And so on, combining each first name with each last name.

But look, why not have JavaScript do the repetitive work? We'll use nested *for* statements.

```
var firstNames = ["BlueRay ", "Upchuck ", "Lojack ", "Gizmo ", "Do-Rag "];
var lastNames = ["Zzz", "Burp", "Dogbone", "Droop"];
var fullNames = [];
for (var i = 0; i < firstNames.length; i++) {
  for (var j = 0; j < lastNames.length; j++) {
    fullNames.push(firstNames[i] + lastNames[j]);
  }
}
```

Things to think about:

- The inner loop runs a complete cycle of iterations on each iteration of the outer loop. If the outer loop counter is **`i`** and the inner loop counter is **`j`**, **`j`** will loop through 0, 1, 2, and all the way to the end while **`i`** is on 0. Then **`i`** will increment to 1, and **`j`** will loop through all of its values again. The outer loop is the minute hand of a clock. The inner loop is the second hand.
- You can have as many levels of nesting as you like.
- A nested loop is indented 2 spaces beyond its outer loop.

Find the interactive coding exercises for this chapter at http://www.ASmarterWayToLearn.com/js/20.html