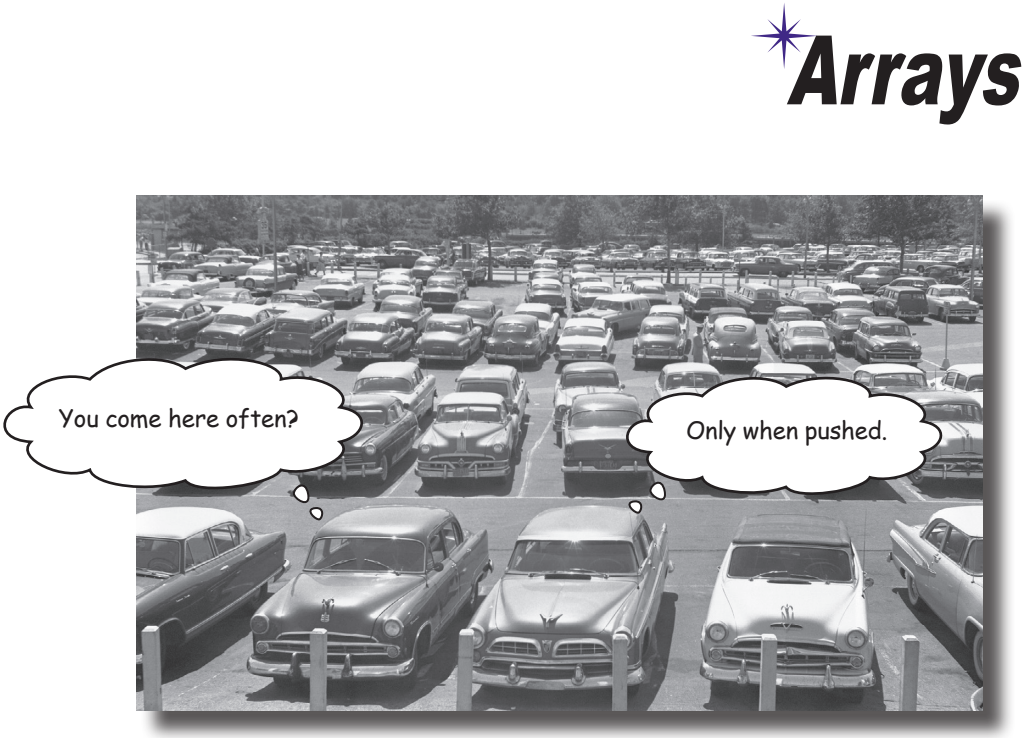
4 putting some order in your data

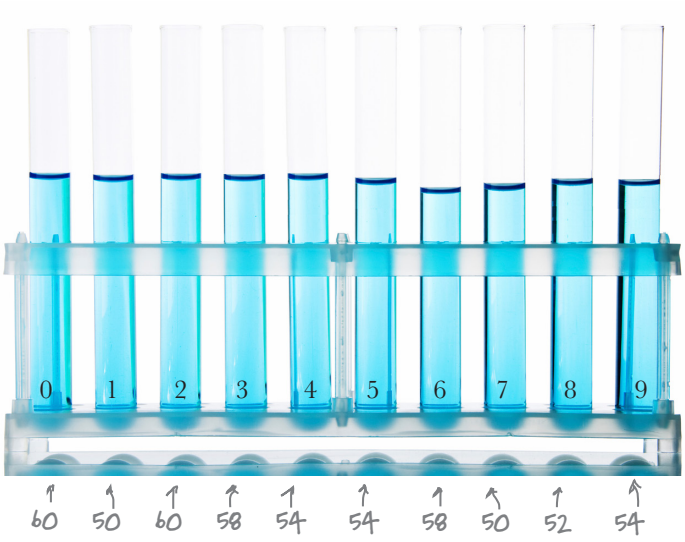


There's more to JavaScript than numbers, strings and booleans. So far you've been writing JavaScript code with **primitives**—simple strings, numbers and booleans, like "Fido", 23, and true. And you can do a lot with primitive types, but at some point you've got to deal with **more data**. Say, all the items in a shopping cart, or all the songs in a playlist, or a set of stars and their apparent magnitude, or an entire product catalog. For that we need a little more *ummph*. The type of choice for this kind of ordered data is a JavaScript **array**, and in this chapter we're going to walk through how to put your data into an array, how to pass it around and how to operate on it. We'll be looking at a few other ways to **structure your data** in later chapters but let's get started with arrays.

Can you help Bubbles-R-Us?

Meet the Bubbles-R-Us company. Their tireless research makes sure bubble wands & machines everywhere blow the best bubbles. Today they're testing the "bubble factor" of several variants of their new bubble solution; that is, they're testing how many bubbles a given solution can make. Here's their data:

Each bubble solution was tested for the number of bubbles it can create.



Each test tube is labelled 0 to 9 and contains a slightly different bubble solution.

And here's the bubble factor score for each solution.

Of course you want to get all this data into JavaScript so you can write code to help analyze it. But that's a lot of values. How are you going to construct your code to handle all these values?

How to represent multiple values in JavaScript

You know how to represent single values like strings, numbers and booleans with JavaScript, but how do you represent *multiple* values, like all the bubble factor scores from the ten bubble solutions? To do that we use JavaScript *arrays*. An array is a JavaScript type that can hold many values. Here's a JavaScript array that holds all the bubble factor scores:

```
var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54];
```

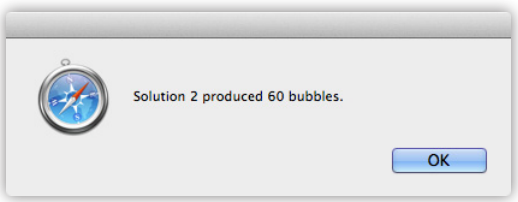
Here's all ten values, grouped together into an array, and assigned to the scores variable.

You can treat all the values as a whole, or you can access the individual scores when you need to. Check this out:

To access an item of the array we use this syntax: the variable name of the array followed by the index of the item, surrounded by square brackets.

Notice that arrays are zero-based. So the first bubble solution is solution #0 and has the score in scores[0], and likewise, the third bubble solution is solution #2 and has the score in scores[2].

```
var solution2 = scores[2];
alert("Solution 2 produced " + solution2 + " bubbles.");
```



One of the Bubbles-R-Us bubbleologists.

How arrays work

Before we get on to helping Bubbles-R-Us, let's make sure we've got arrays down. As we said, you can use arrays to store *multiple* values (unlike variables that hold just one value, like a number or a string). Most often you'll use arrays when you want to group together similar things, like bubble factor scores, ice cream flavors, daytime temperatures or even the answers to a set of true/false questions. Once you have a bunch of values you want to group together, you can create an array that holds them, and then access those values in the array whenever you need them.

How to create an array

Let's say you wanted to create an array that holds ice cream flavors. Here's how you'd do that:

```
var flavors = ["vanilla", "butterscotch", "lavender", "chocolate", "cookie dough"];
```

Notice that each item in the array is separated by a comma.

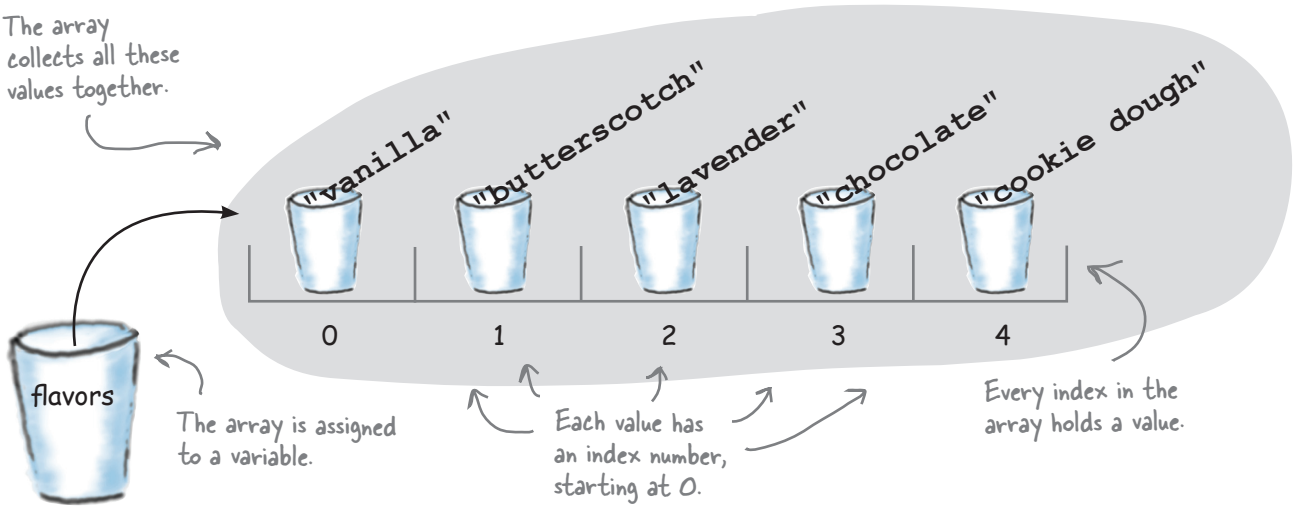
Let's assign the array to a variable named flavors.

To begin the array, use the [character...

and then list each item of the array...

... and end the array with the] character.

When you create an array, each item is placed at a location, or *index*, in the array. With the flavors array, the first item, "vanilla", is at index 0, the second, "butterscotch", is at index 1, and so on. Here's a way to think about an array:



How to access an array item

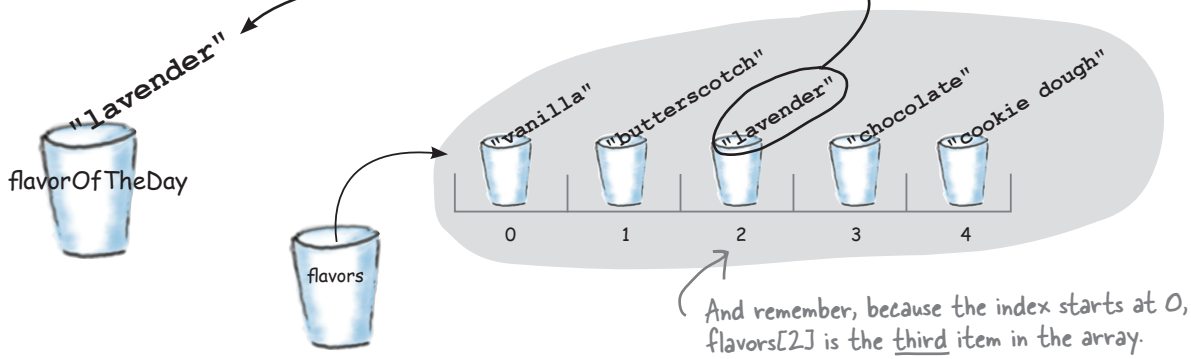
Each item in the array has its own index, and that's your key to both accessing and changing the values in an array. To access an item just follow the array variable name with an index, surrounded by square brackets. You can use that notation anywhere you'd use a variable:

```
var flavorOfTheDay = flavors[2];
```

This evaluates to the value of the flavors array at index 2 (which is "lavender"), which is then assigned to the variable flavorOfTheDay.

To get an element from an array, you need both the name of the array, and the index of the value you want.

flavorOfTheDay is assigned the value in flavors[2].



And remember, because the index starts at 0, flavors[2] is the third item in the array.

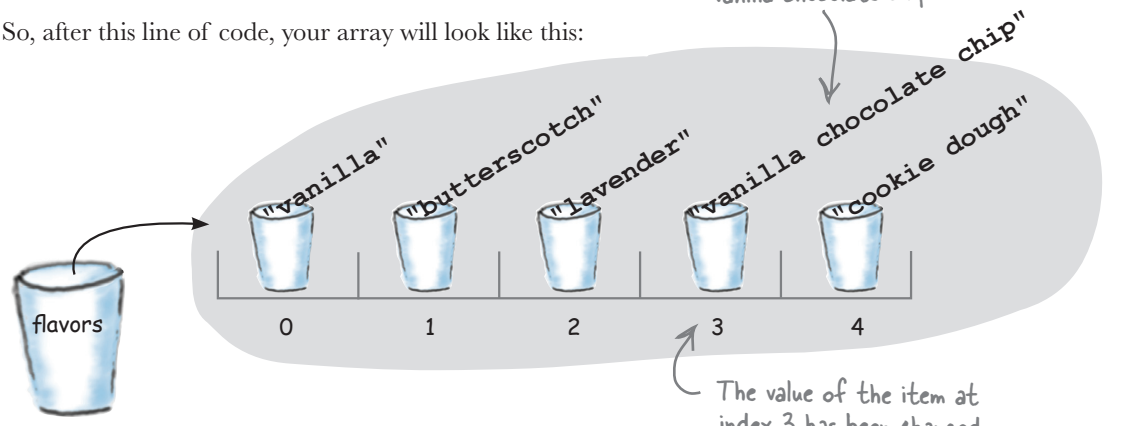
Updating a value in the array

You can also use the array index to change a value in an array:

```
flavors[3] = "vanilla chocolate chip";
```

This sets the value of the item at index 3 (previously "chocolate") to a new value, "vanilla chocolate chip".

So, after this line of code, your array will look like this:



The value of the item at index 3 has been changed.

How big is that array anyway?

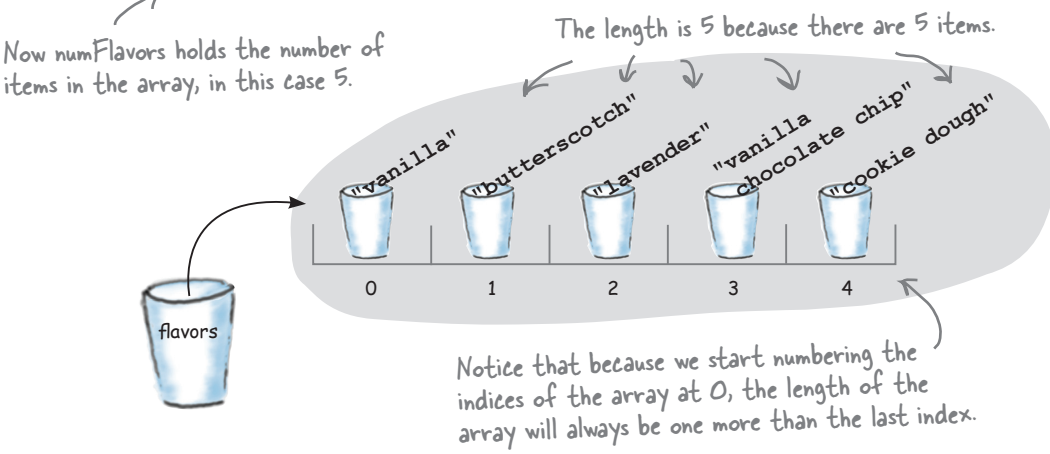
Say someone hands you a nice big array with important data in it. You know what's in it, but you probably won't know exactly how big it is. Luckily, every array comes with its own property, `length`. We'll talk more about properties and how they work in the next chapter, but for now, a property is just a value associated with an array. Here's how you use the `length` property:

Every array has a property, length, that contains the number of items currently in the array.

To get the length of an array, you use the name of the array, then a ".", and then length.

```
var numFlavors = flavors.length;
```

Now numFlavors holds the number of items in the array, in this case 5.



Sharpen your pencil

The products array below holds the Jenn and Berry's ice cream flavors. The ice cream flavors were added to this array in the order of their creation. Finish the code to determine the *most recent* ice cream flavor they created.

```
var products = ["Choo Choo Chocolate", "Icy Mint", "Cake Batter", "Bubblegum"];
var last = _________________;
var recent = products[last];
```

Try my new Phrase-o-Matic and you'll be a slick talker just like the boss or those guys in marketing.

You didn't think our serious business application from Chapter 1 was serious enough? Fine. Try this one, if you need something to show the boss.

Check out this code for the hot new Phrase-o-Matic app and see if you can figure out what it does before you go on...

```
<!doctype html>
<html lang="en">
<head>
  <title>Phrase-o-matic</title>
  <meta charset="utf-8">
  <script>
    function makePhrases() {
      var words1 = ["24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win"];
      var words2 = ["empowered", "value-added", "oriented", "focused", "aligned"];
      var words3 = ["process", "solution", "tipping-point", "strategy", "vision"];

      var rand1 = Math.floor(Math.random() * words1.length);
      var rand2 = Math.floor(Math.random() * words2.length);
      var rand3 = Math.floor(Math.random() * words3.length);

      var phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
      alert(phrase);
    }
    makePhrases();
  </script>
</head>
<body></body>
</html>
```

The Phrase-O-Matic

We hope you figured out this code is the perfect tool for creating your next start-up marketing slogan. It has created winners like "Win-win value-added solution" and "24/7 empowered process" in the past and we have high hopes for more winners in the future. Let's see how this thing really works:

1. First, we define the `makePhrases` function, which we can call as many times as we want to generate the phrases we want:

We're defining a function named makePhrases, that we can call later.

```
function makePhrases() {
}
makePhrases();
```

All the code for makePhrases goes here, we'll get to it in a sec...

We call makePhrases once here, but we could call it multiple times if we want more than one phrase.

2. With that out of the way we can write the code for the `makePhrases` function. Let's start by setting up three arrays. Each will hold words that we'll use to create the phrases. In the next step, we'll pick one word at random from each array to make a three word phrase.

We create a variable named words1, that we can use for the first array.

```
var words1 = ["24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win"];
```

We're putting five strings in the array. Feel free to change these to the latest buzzwords out there.

```
var words2 = ["empowered", "value-added", "oriented", "focused", "aligned"];
var words3 = ["process", "solution", "tipping-point", "strategy", "vision"];
```

And here are two more arrays of words, assigned to two new variables, words2 and words3.

3. Now we generate three random numbers, one for each of the three random words we want to pick to make a phrase. Remember from Chapter 2 that Math.random generates a number between 0 and 1 (not including 1). If we multiply that by the length of the array, and use Math.floor to truncate the number, we get a number between 0 and one less than the length of the array.

```
var rand1 = Math.floor(Math.random() * words1.length);
var rand2 = Math.floor(Math.random() * words2.length);
var rand3 = Math.floor(Math.random() * words3.length);
```

rand1 will be a number between 0 and the last index of the words1 array.

And likewise for rand2, and rand3.

4. Now we create the slick marketing phrase by taking each randomly chosen word and concatenating them all together, with a nice space in between for readability:

We define another variable to hold the phrase.

We use each random number to index into the word arrays...

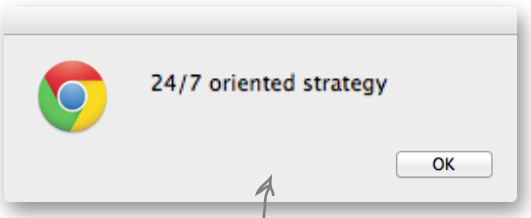
```
var phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
```

5. We're almost done; we have the phrase, now we just have to display it. We're going to use `alert` as usual.

```
alert(phrase);
```

6. Okay, finish that last line of code, have one more look over it all and feel that sense of accomplishment before you load it into your browser. Give it a test drive and enjoy the phrases.

Here's what ours looks like!



Just reload the page for endless start-up possibilities (okay, not endless, but work with us here, we're trying to make this simple code exciting!).

there are no Dumb Questions

Q: Does the order of items in an array matter?

A: Most of the time, yes, but it depends. In the Bubbles-R-Us scores array, the ordering matters a lot, because the index of the score in the array tells us which bubble solution got that score—bubble solution 0 got score 60, and that score is stored at index 0. If we mixed up the scores in the array, then we'd ruin the experiment! However, in other cases, the order may not matter. For instance, if you're using an array just to keep a list of randomly selected words and you don't care about the order, then it doesn't matter which order they're in the array. But, if you later decide you want the words to be in alphabetical order, then the order will matter. So it really depends on how you're using the array. You'll probably find that ordering matters more often than not when you use an array.

Q: How many things can you put into an array?

A: Theoretically, as many as you want. Practically, however, the number is limited by the memory on your computer. Each array item takes up a little bit of space in memory. Remember that JavaScript runs in a browser, and that browser is one of many programs running on your computer. If you keep adding items to an array, eventually you'll run out of memory space. However, depending on the kind of items you're putting in your array, the maximum number of items you can put into an array is probably in the many thousands, if not millions, which you're unlikely to need most of the time. And keep in mind that the more items you have the slower your program will run, so you'll want to limit your arrays to reasonable sizes—say a few hundred—most of the time.

Q: Can you have an empty array?

A: You can, and in fact, you'll see an example of using an empty array shortly. To create an empty array, just write:

```
var emptyArray = [ ];
```

If you start with an empty array, you can add things to it later.

Q: So far we've seen strings and numbers in an array; can you put other things in arrays too?

A: You can; in fact, you can put just about any value you'll find in JavaScript in an array, including numbers, strings, booleans, other arrays, and even objects (we'll get to this later).

Q: Do all the values in an array have to be the same type?

A: No they don't; although typically we *do* make the values all of the same type. Unlike many other languages, there is no requirement in JavaScript that all the values in an array be of the same type. However, if you mix up the types of the values in an array, you need to be extra careful when using those values. Here's why: let's say you have an array with the values [1, 2, "fido", 4, 5]. If you then write code that checks to see if the values in the array are greater than, say, 2, what happens when you check to see if "fido" is greater than 2? To make sure you aren't doing something that doesn't make sense, you'd have to check the type of each of the values before you used it in the rest of your code. It's certainly possible to do this (and we'll see later in the book how), but in general, it's a lot easier and safer if you just use the same type for all the values in your arrays.

Q: What happens if you try to access an array with an index that is too big or too small (like less than 0)?

A: If you have an array, like:

```
var a = [1, 2, 3];
```

and you try to access a[10] or a[-1], in either case, you'll get the result undefined. So, you'll either want to make sure you're using only valid indices to access items in your array, or you'll need to check that the value you get back is not undefined.

Q: So, I can see how to get the first item in an array using index 0. But how would I get the last item in an array? Do I always have to know precisely how many items are in my array?

A: You can use the length property to get the last item of an array. You know that length is always one greater than the last index of the array, right? So, to get the last item in the array, you can write:

```
myArray[myArray.length - 1];
```

JavaScript gets the length of the array, subtracts one from it, and then gets the item at that index number. So if your array has 10 items, it will get the item at index 9, which is exactly what you want. You'll use this trick all the time to get the last item in an array when you don't know exactly how many items are in it.

Meanwhile, back at Bubbles-R-Us...

Hey, glad you guys are here. We just got a lot of new bubble tests run. Check out all the new bubble scores! I really need some help understanding this data. I'd love for you to code up what I sketched below.

The Bubbles-R-Us CEO

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];
```

What we need to build.

New bubble scores.

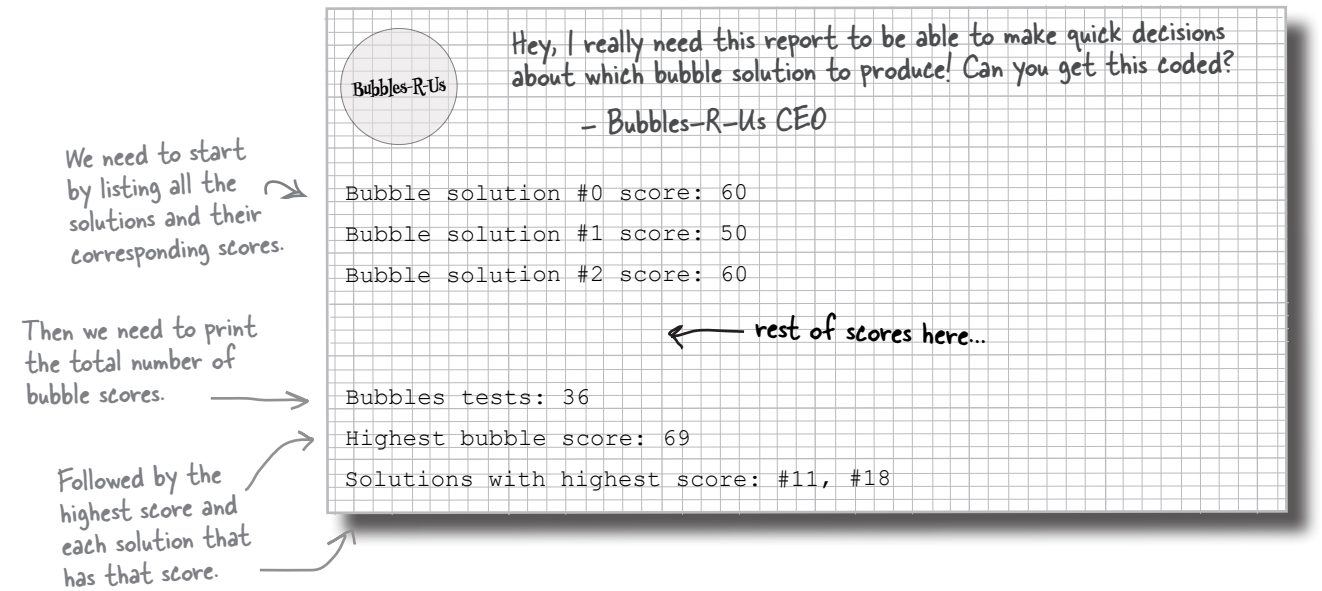
Bubbles-R-Us

Hey, I really need this report to be able to make quick decisions about which bubble solution to produce! Can you get this coded?

– Bubbles-R-Us CEO

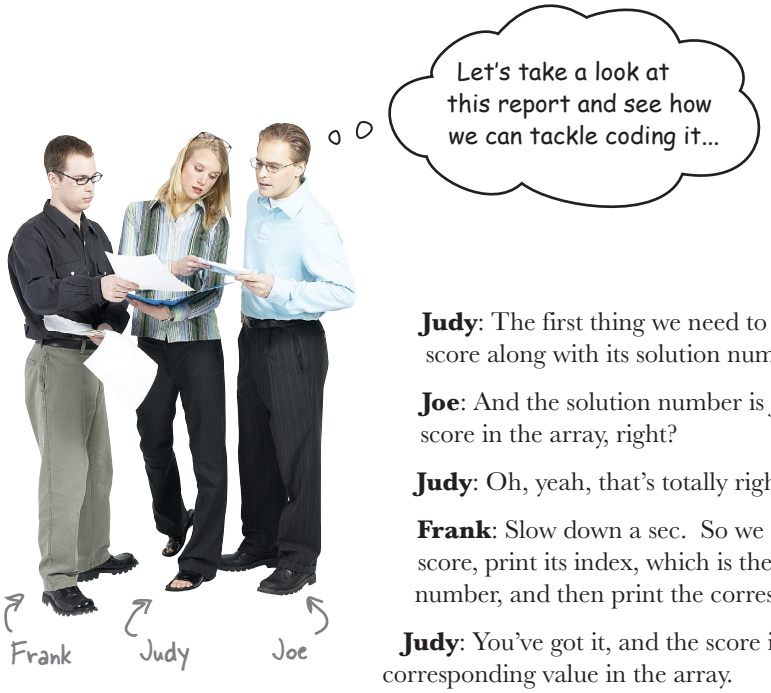
```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60

                          <-- rest of scores here...

Bubbles tests: 36
Highest bubble score: 69
Solutions with highest score: #11, #18
```

Let's take a closer look at what the CEO is looking for:



Take some time to sketch out your ideas of how you'd create this little bubble score report. Take each item in the report separately and think of how you'd break it down and generate the right output. Make your notes here.

Cubicle Conversation



Judy: The first thing we need to do is display every score along with its solution number.

Joe: And the solution number is just the index of the score in the array, right?

Judy: Oh, yeah, that's totally right.

Frank: Slow down a sec. So we need to take each score, print its index, which is the bubble solution number, and then print the corresponding score.

Judy: You've got it, and the score is just the corresponding value in the array.

Joe: So, for bubble solution #10, its score is just `scores[10]`.

Judy: Right.

Frank: Okay, but there are a lot of scores. How do we write code to output all of them?

Judy: Iteration, my friend.

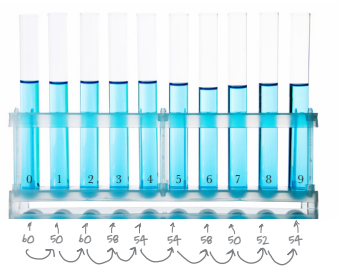
Frank: Oh, you mean like a while loop?

Judy: Right, we loop through all the values from zero to the length... oh, I mean the length minus one of course.

Joe: This is starting to sound very doable. Let's write some code; I think we know what we're doing.

Judy: That works for me! Let's do it, and then we'll come back to the rest of the report.

How to iterate over an array



Your goal is to produce some output that looks like this:

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
.
.
.
Bubble solution #35 score: 44
```

Scores 3 through 34 will be here... we're saving some trees (or bits depending on which version of the book you have).

We'll do that by outputting the score at index zero, and then we'll do the same for index one, two, three and so on, until we reach the last index in the array. You already know how to use a while loop; let's see how we can use that to output all the scores:

And then we'll show you a better way in a sec...

```
var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

var output;

var i = 0;

while (i < scores.length) {

  output = "Bubble solution #" + i + " score: " + scores[i];

  console.log(output);

  i = i + 1;

}
```

We're using this variable in the loop below to create a string to output.

Create a variable to keep track of the current index.

And keep looping while our index is less than the length of the array.

Then create a string to use as a line of output that includes the bubble solution number (which is just the array index) and the score.

Then we'll use console.log to output the string.

And finally, increment the index by one before looping again.

Code Magnets

We've got code for testing to see which ice cream flavors have bubblegum pieces in them. We had all the code nicely laid out on our fridge using fridge magnets, but the magnets fell on the floor. It's your job to put them back together. Be careful; a few extra magnets got mixed in. Check your answer at the end of the chapter before you go on.

```
while (i < hasBubbleGum.length)
```

```
i = i + 2;
```

```
{
```

```
}
```

```
i = i + 1;
```

```
}
```

```
var i = 0;
```

```
{
```

```
{
```

```
if (hasBubbleGum[i])
```

```
while (i > hasBubbleGum.length)
```

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
```

```
var hasBubbleGum = [false,
                    false,
                    false,
                    true];
```

```
console.log(products[i] +
    " contains bubble gum");
```

Rearrange the magnets here.

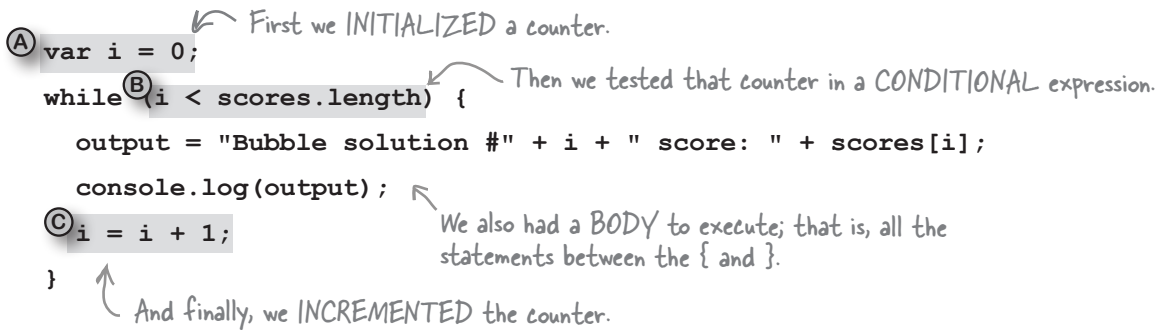
Here's the output we're expecting.

JavaScript console

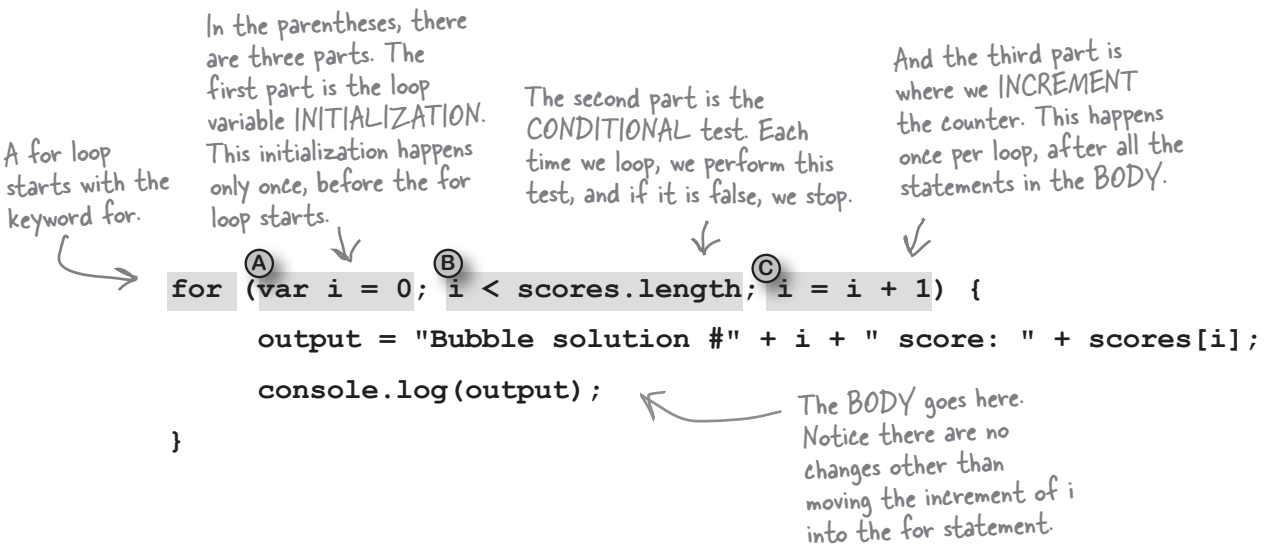
```
Bubblegum contains bubble gum
```

But wait, there's a better way to iterate over an array

We should really apologize. We can't believe it's already Chapter 4 and we haven't even introduced you to the **for loop**. Think of the for loop as the while loop's cousin. The two basically do the same thing, except the for loop is usually a little more convenient to use. Check out the while loop we just used and we'll see how that maps into a for loop.



Now let's look at how the for loop makes all that so much easier:

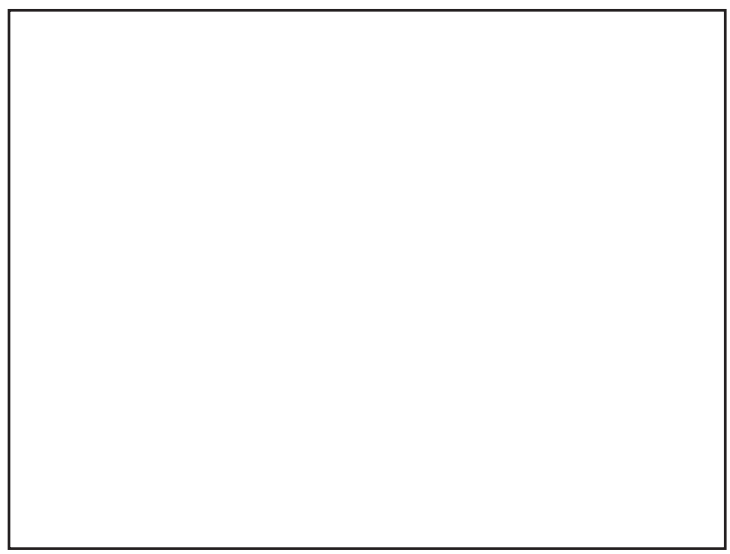


Sharpen your pencil

Rewrite your fridge magnet code (from two pages back) so that it uses a for loop instead of a while loop. If you need a hint, refer to each piece of the while loop on the previous page and see how it maps to the corresponding location in the for loop.

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];

var hasBubbleGum = [false,
                    false,
                    false,
                    true];

var i = 0;
while (i < hasBubbleGum.length)
{
  if (hasBubbleGum[i])
  {
    console.log(products[i] +
        " contains bubble gum");
  }
  i = i + 1;
}
```

Your code goes here.



We've got the standard HTML stuff here for a web page. We don't need much; just enough to create a script.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    var scores = [60, 50, 60, 58, 54, 54,
                  58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51,
                  69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18,
                  41, 53, 55, 61, 51, 44];

    var output;

    for (var i = 0; i < scores.length; i = i + 1) {

        output = "Bubble solution #" + i +
                    " score: " + scores[i];

        console.log(output);

    }

  </script>
</head>
<body></body>
</html>
```

Here's our bubble scores array.

Here's the for loop we're using to iterate through all the bubble solution scores.

Each time through the loop, we create a string with the value of i, which is the bubble solution number, and scores[i], which is the score that bubble solution got.

(Also notice we split the string up across two lines here. That's okay as long as you don't create a new line in between the quotes that delimit a string. Here, we did it after a concatenation operator (+), so it's okay. Be careful to type it in exactly as you see here.)

Then we display the string in the console. And that's it! Time to run this report.

Test drive the bubble report

Save this file as "bubbles.html" and load it into your browser. Make sure you've got the console visible (you might need to reload the page if you activate the console after you load the page), and check out the brilliant report you just generated for the Bubbles-R-Us CEO.

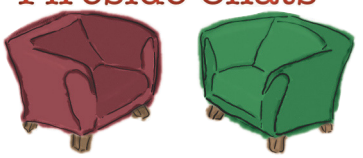
JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
Bubble solution #3 score: 58
Bubble solution #4 score: 54
Bubble solution #5 score: 54
Bubble solution #6 score: 58
Bubble solution #7 score: 50
Bubble solution #8 score: 52
Bubble solution #9 score: 54
Bubble solution #10 score: 48
Bubble solution #11 score: 69
Bubble solution #12 score: 34
Bubble solution #13 score: 55
Bubble solution #14 score: 51
Bubble solution #15 score: 52
Bubble solution #16 score: 44
Bubble solution #17 score: 51
Bubble solution #18 score: 69
Bubble solution #19 score: 64
Bubble solution #20 score: 66
Bubble solution #21 score: 55
Bubble solution #22 score: 52
Bubble solution #23 score: 61
Bubble solution #24 score: 46
Bubble solution #25 score: 31
Bubble solution #26 score: 57
Bubble solution #27 score: 52
Bubble solution #28 score: 44
Bubble solution #29 score: 18
Bubble solution #30 score: 41
Bubble solution #31 score: 53
Bubble solution #32 score: 55
Bubble solution #33 score: 61
Bubble solution #34 score: 51
Bubble solution #35 score: 44
```

Just what the CEO ordered.

It's nice to see all the bubble scores in a report, but it's still hard to find the highest scores. We need to work on the rest of the report requirements to make it a little easier to find the winner.

Fireside Chats



Tonight's talk: **The while and for loop answer the question "Who's more important?"**

The WHILE loop

What, are you kidding me? Hello? I'm the *general* looping construct in JavaScript. I'm not married to looping with a silly counter. I can be used with any type of conditional. Did anyone notice I was taught first in this book?

The FOR loop

I don't appreciate that tone.

And that's another thing, have you noticed that the FOR loop has no sense of humor? I mean if we all had to do skull-numbing iteration all day I guess we'd all be that way.

Cute. But have you noticed that nine times out of ten, coders use FOR loops?

Oh, I don't think that could possibly be true.

Not to mention, doing iteration over, say, an array that has a fixed number of items with a WHILE loop is just a bad, clumsy practice.

This book just showed that FOR and WHILE loops are pretty much the same thing, so how could that be?

Ah, so you admit we're more equal than you let on huh?

I'll tell you why...

The WHILE loop

Well, isn't that nice and neat of you. Hey, most of the iteration I see doesn't even include counters; it's stuff like:

```
while (answer != "forty-two")
```

try that with a FOR loop!

Hah, I can't believe that even works.

Lipstick on a pig.

Not only better, prettier.

The FOR loop

When you use a WHILE loop you have to initialize your counter and increment your counter in separate statements. If, after lots of code changes, you accidentally moved or deleted one of these statements, well, then things could get ugly. But with a FOR loop, everything is packaged right in the FOR statement for all to see and with no chance of things getting changed or lost.

Okay:

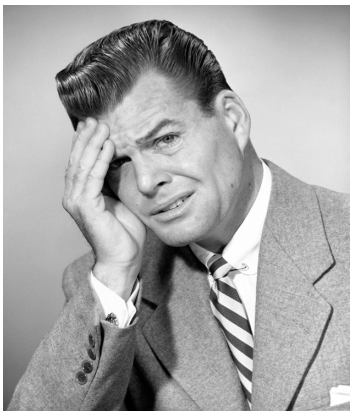
```
for (;answer != "forty-two";)
```

Oh, it does.

So that's all you got? You're better when you've got a general conditional?

Oh, I didn't realize this was a beauty contest as well.

It's that time again.... Can we talk about your verbosity?



You've been writing lots of code that looks like this:

Assume myImportantCounter contains a number, like 0.

Here we're taking the variable and incrementing it by one.

```
myImportantCounter = myImportantCounter + 1;
```

After this statement completes, myImportantCounter is one greater than before.

In fact, this statement is so common there's a shortcut for it in JavaScript. It's called the post-increment operator, and despite its fancy name, it is quite simple. Using the post-increment operator, we can replace the above line of code with this:

Just add "++" to the variable name.

```
myImportantCounter++;
```

After this statement completes, myImportantCounter is one greater than before.

Of course it just wouldn't feel right if there wasn't a post-decrement operator as well. You can use the post-decrement operator on a variable to reduce its value by one. Like this:

Just add "--" to the variable name.

```
myImportantCounter--;
```

After this statement completes, myImportantCounter is one less than before.

And why are we telling you this now? Because it's commonly used with `for` statements. Let's clean up our code a little using the post-increment operator…

Redoing the for loop with the post-increment operator

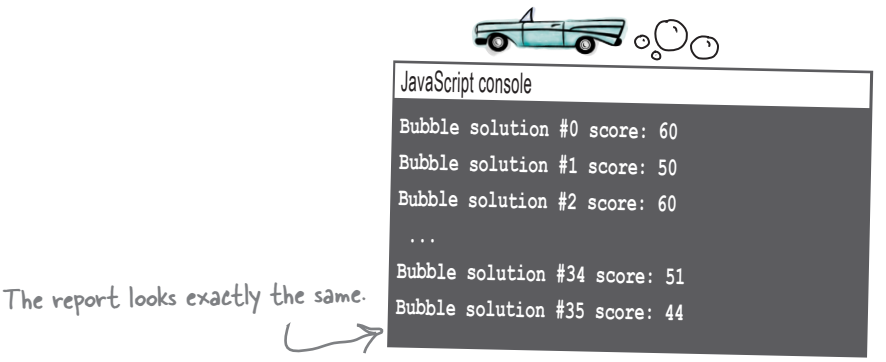
Let's do a quick rewrite and test to make sure the code works the same as before:

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

for (var i = 0; i < scores.length; i++) {

    var output = "Bubble solution #" + i +
                 " score: " + scores[i];

    console.log(output);

}
```

All we've done is update where we increment the loop variable with the post-increment operator.

Quick test drive

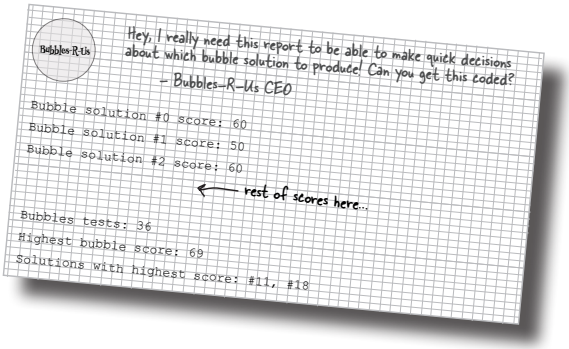
Time to do a quick test drive to make sure the change to use the post-increment operator works. Save your file, "bubbles.html", and reload. You should see the same report you saw before.



The report looks exactly the same.

We're saving a few trees and not showing all the bubble solution scores, but they are all there.

Cubicle Conversation Continued...



Judy: Right, and the first thing we need to do is determine the total number of bubble tests. That's easy; it's just the length of the scores array.

Joe: Oh, right. We've got to find the highest score too, and then the solutions that have the highest score.

Judy: Yeah, that last one is going to be the toughest. Let's work out finding the highest score first.

Joe: Sounds like a good place to start.

Judy: To do that I think we just need to maintain a highest score variable that keeps track as we interate through the array. Here, let me write some pseudocode:

```
DECLARE a variable highScore and set to zero.
FOR: var i=0; i < scores.length; i++
    DISPLAY the bubble solution score[i]
    IF scores[i] > highScore
        SET highScore = scores[i];
    END IF
END FOR
DISPLAY highScore
```

Add a variable to hold the high score.

Check each time through the loop to see if we have a higher score, and if so that's our new high score.

After the loop we just display the high score.

Joe: Oh nice; you did it with just a few lines added to our existing code.

Judy: Each time through the array we look to see if the current score is greater than `highScore`, and if so, that's our new high score. Then, after the loop ends we just display the high score.

Sharpen your pencil

Go ahead and implement the pseudocode on the previous page to find the highest score by filling in the blanks in the code below. Once you're done, give it a try in the browser by updating the code in "bubbles.html" and reloading the page. Check the results in the console, and fill in the blanks in our console display below with the number of bubble tests and the highest score. Check your answer at the end of the chapter before you go on.

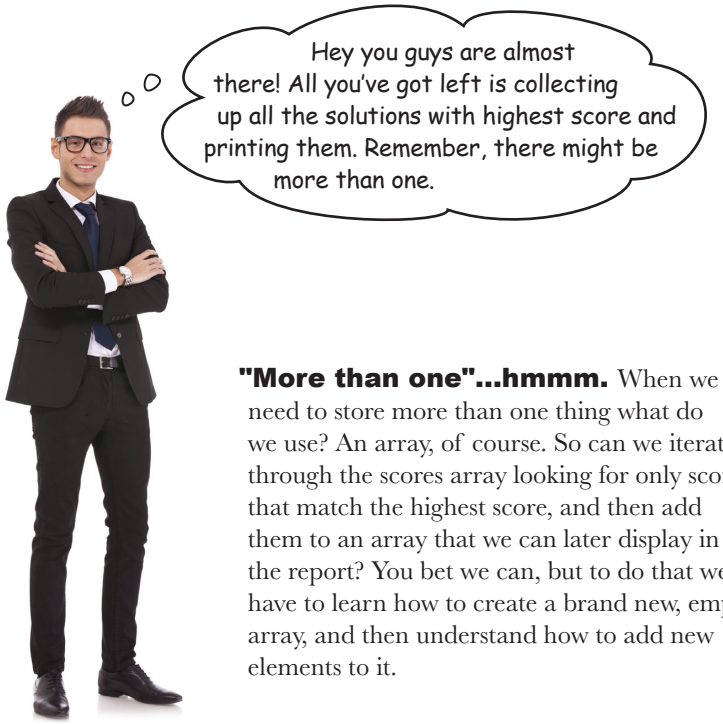
```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

var highScore = _____;
var output;
for (var i = 0; i < scores.length; i++) {
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (_____________ > highScore) {
        _____________ = scores[i];
    }
}
console.log("Bubbles tests: " + _____________);
console.log("Highest bubble score: " + _____________);
```

Fill in the blanks to complete the code here...

... and then fill in the blanks showing the output you get in the console.

JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: _____
Highest bubble score: _____
```



"More than one"...hmmm. When we need to store more than one thing what do we use? An array, of course. So can we iterate through the scores array looking for only scores that match the highest score, and then add them to an array that we can later display in the report? You bet we can, but to do that we'll have to learn how to create a brand new, empty array, and then understand how to add new elements to it.

Remember here's what we have left.

Bubbles-R-Us

Hey, I really need this report to be able to make quick decisions about which bubble solution to produce! Can you get this coded?

– Bubbles-R-Us CEO

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60

                          <-- rest of scores here...

Bubbles tests: 36
Highest bubble score: 69
Solutions with highest score: 11, 18
```

Creating an array from scratch (and adding to it)

Before we take on finishing this code, let's get a sense for how to create a new array, and how to add new items to it. You already know how to create an array with values, like this:

```
var genres = ["80s", "90s", "Electronic", "Folk"];
```

This is called an array literal, because we're literally writing out what goes in the array.

But you can also omit the initial items and just create an empty array:

```
var genres = [];
```

A new array, all ready to go with no items and a length of zero.

This is an array literal too, it just doesn't have anything in it (yet).

And you already know how to add new values to an array. To do that you just assign a value to an item at an index, like this:

```
var genres = [];
genres[0] = "Rockabilly";
genres[1] = "Ambient";
var size = genres.length;
```

A new array item is created and it holds the string "Rockabilly".

And a second array item is created that holds the string "Ambient".

And here size holds the value 2, the length of the array.

Now when adding new items you have to be careful about which index you're adding. Otherwise you'll create a sparse array, which is an array with "holes" in it (like an array with values at 0 and 2, but no value at 1). Having a sparse array isn't necessarily a bad thing, but it does require special attention. For now, there's another way to add new items without worrying about the index, and that's `push`. Here's how it works:

```
var genres = [];
genres.push("Rockabilly");
genres.push("Ambient");
var size = genres.length;
```

Creates a new item in the next available index (which happens to be 0) and sets its value to "Rockabilly".

Creates another new item in the next open index (in this case, 1) and sets the value to "Ambient".

there are no Dumb Questions

Q: The for statement contains a variable declaration and initialization in the first part of the statement. You said we should put our variable declarations at the top. So, what gives?

A: Yes, putting your variable declarations at the top (of your file, if they are global, or of your function if they are local) is a good practice. However, there are times when it makes sense to declare a variable right where you're going to use it, and a for statement is one of those times. Typically, you use a loop variable, like i, just for iterating, and once the loop is done, you're done with that variable. Now, you might use i later in your code, of course, but typically you won't. So, in this case, just declaring it right in the for statement keeps things tidy.

Q: What does the syntax myarray.push(value) actually mean?

A: Well, we've been keeping a little secret from you: in JavaScript, an array is actually a special kind of object. As you'll learn in the next chapter, an object can have functions associated with it that act on the object. So, think of push as a function that can act on myarray. In this case, what that function does is add a new item to the array, the item that you pass as an argument to push. So, if you write

```
genres.push("Metal");
```

you're calling the function push and passing it a string argument, "Metal". The push function takes that argument and adds it as a new value on the end of the genres array. When you see `myarray.push(value)` just think, "I'm pushing a new value on the end of my array."

Q: Can you say a little more about what a sparse array is?

A: A sparse array is just an array that has values at only a few indices and no values in between. You can create a sparse array easily, like this:

```
var sparseArray = [ ];
sparseArray[0] = true;
spraseArray[100] = true;
```

In this example, the sparseArray has only two values, both true, at indices 0 and 100.The values at all the other indices are undefined. The length of the array is 101 even though there are only two values.

Q: Say I have an array of length 10, and I add a new item at index 10000, what happens with indices 10 through 9999?

A: All those array indices get the value undefined. If you remember, undefined is the value assigned to a variable that you haven't initialized. So, think of this as if you're creating 9989 variables, but not initializing them. Remember that all those variables take up memory in your computer, even if they don't have a value, so make sure you have a good reason to create a sparse array.

Q: So, if I'm iterating through an array, and some of the values are undefined, should I check to make sure before I use them?

A: If you think your array might be sparse, or even have just one undefined value in it, then yes, you should probably check to make sure that the value at an array index is not undefined before you use it. If all you're doing is displaying the value in the console, then it's no big deal, but it's much more likely that you'll actually want to use that value somehow, perhaps in a calculation of some kind. In that case, if you try to use undefined, you might get an error, or at the very least, some unexpected behavior. To check for undefined, just write:

```
if (myarray[i] == undefined) {
  ...
}
```

Notice there are no quotes around undefined (because it's not a string, it's a value).

Q: All the arrays we've created so far have been literal. Is there another way to create an array?

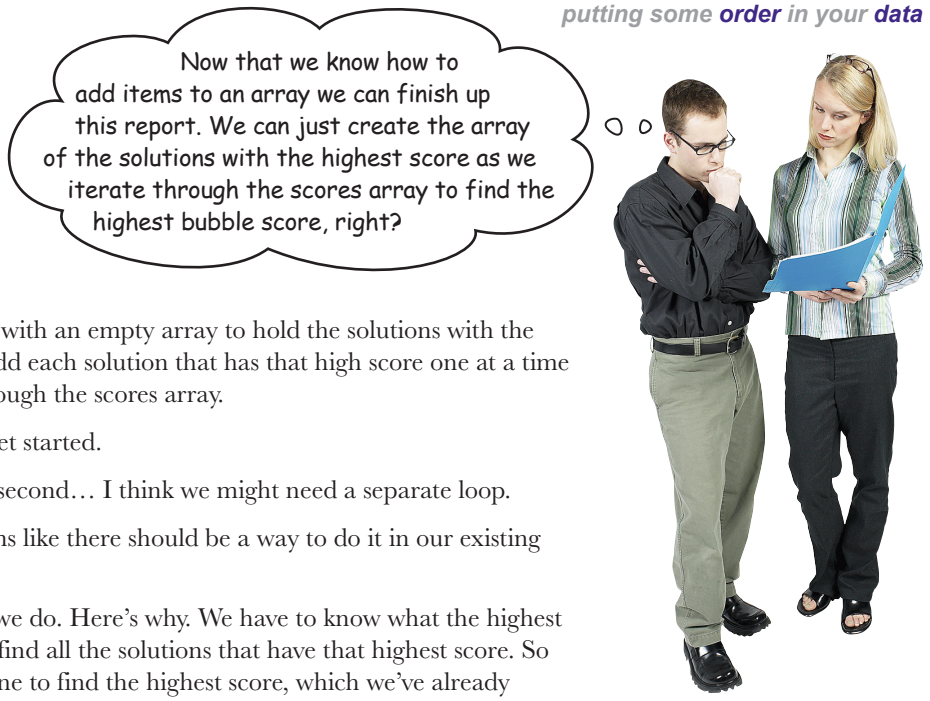
A: Yes. You may have seen the syntax:

```
var myarray = new Array(3);
```

What this does is create a new array, with three empty spots in it (that is, an array with length 3, but no values yet). Then you can fill them, just like you normally would, by providing values for myarray at indices 0, 1, and 2. Until you add values yourself, the values in myarray are undefined.

An array created this way is just the same as an array literal, and in practice, you'll find yourself using the literal syntax more often, and that's what we'll tend to use in the rest of the book.

And don't worry about the details of the syntax above for now (like "new" and why Array is capitalized); we'll get to all that later!



Judy: Yes, we'll start with an empty array to hold the solutions with the highest scores, and add each solution that has that high score one at a time to it as we iterate through the scores array.

Frank: Great, let's get started.

Judy: But hold on a second… I think we might need a separate loop.

Frank: We do? Seems like there should be a way to do it in our existing loop.

Judy: Yup, I'm sure we do. Here's why. We have to know what the highest score is *before* we can find all the solutions that have that highest score. So we need two loops: one to find the highest score, which we've already written, and then a second one to find all the solutions that have that score.

Frank: Oh, I see. And in the second loop, we'll compare each score to the highest score, and if it matches, we'll add the index of the bubble solution score to the new array we're creating for the solutions with the highest scores.

Judy: Exactly! Let's do it.

Sharpen your pencil

Can you write the loop to find all the scores that match the high score? Give it a shot below before you turn the page to see the solution and give it a test drive.

Remember, the variable highScore has the highest score in it; you can use that in the code below.

```
var bestSolutions = [];
for (var i = 0; i < scores.length; i++) {


}
```

Here's the new array we'll use to store the bubble solutions with the highest score.

Your code here.

Sharpen your pencil Solution

Can you write the loop to find all the scores that match the high score? Here's our solution.

Again, we're starting by creating a new array that will hold all the bubble solutions that match the highest score.

Next, we iterate through the entire scores array, looking for those items with the highest score.

```
var bestSolutions = [];

for (var i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {

        bestSolutions.push(i);

    }
}

console.log("Solutions with the highest score: " + bestSolutions);
```

Each time through the loop, we compare the score at index i with the highScore and if they are equal, then we add that index to the bestSolutions array using push.

And finally, we can display the bubble solutions with the highest scores. Notice we're using console.log to display the bestSolutions array. We could create another loop to display the array items one by one, but, luckily, console.log will do this for us (and, if you look at the output, it also adds commas between the array values!).

Take a look at the code in the Sharpen exercise above. What if you woke up and push no longer existed? Could you rewrite this code without using push? Work that code out here:

Test drive the final report

Go ahead and add the code to generate the bubble solutions with the highest score to your code in "bubbles.html" and run another test drive. All the JavaScript code is shown below:

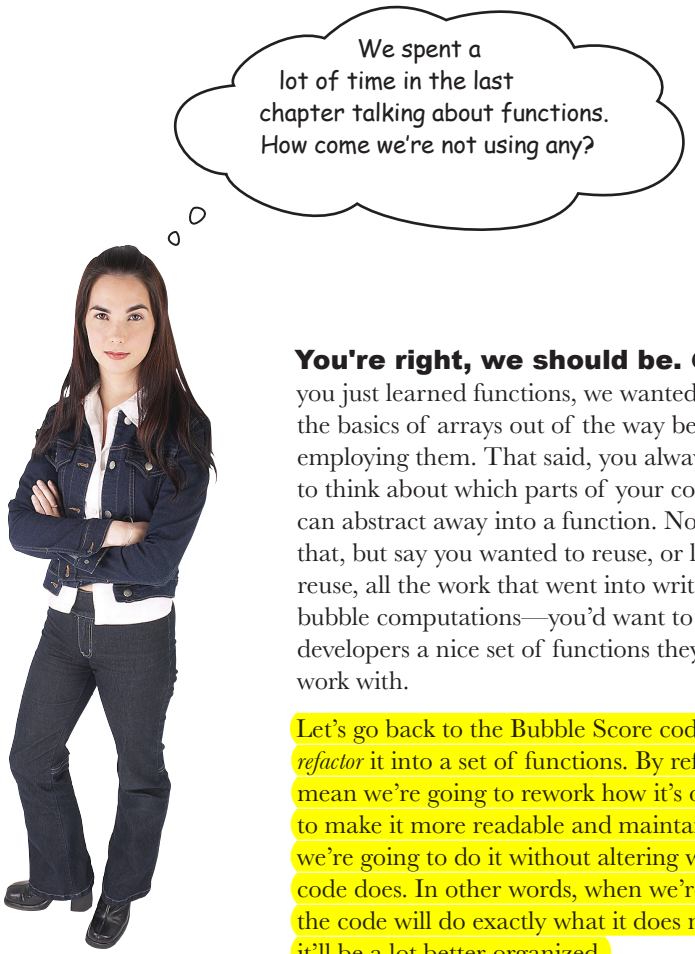
```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

var highScore = 0;
var output;
for (var i = 0; i < scores.length; i++) {
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (scores[i] > highScore) {
        highScore = scores[i];
    }
}
console.log("Bubbles tests: " + scores.length);
console.log("Highest bubble score: " + highScore);

var bestSolutions = [];
for (var i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {
        bestSolutions.push(i);
    }
}
console.log("Solutions with the highest score: " + bestSolutions);
```

And the winners are...

Bubble solutions #11 and #18 both have a high score of 69! So they are the best bubble solutions in this batch of test solutions.

JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
 ...
Bubble solution #34 score: 51
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11,18
```



You're right, we should be. Given you just learned functions, we wanted to get the basics of arrays out of the way before employing them. That said, you always want to think about which parts of your code you can abstract away into a function. Not only that, but say you wanted to reuse, or let others reuse, all the work that went into writing the bubble computations—you'd want to give other developers a nice set of functions they could work with.

Let's go back to the Bubble Score code and *refactor* it into a set of functions. By refactor we mean we're going to rework how it's organized, to make it more readable and maintainable, but we're going to do it without altering what the code does. In other words, when we're done, the code will do exactly what it does now but it'll be a lot better organized.

A quick survey of the code...

Let's get an overview of the code we've written and figure out which pieces we want to abstract into functions:

Here's the Bubbles-R-Us code.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    var scores = [60, 50, 60, 58, 54, 54,
                  58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51,
                  69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18,
                  41, 53, 55, 61, 51, 44];

    var highScore = 0;
    var output;

    for (var i = 0; i < scores.length; i++) {
      output = "Bubble solution #" + i + " score: " + scores[i];
      console.log(output);
      if (scores[i] > highScore) {
        highScore = scores[i];
      }
    }
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    var bestSolutions = [];

    for (var i = 0; i < scores.length; i++) {
      if (scores[i] == highScore) {
        bestSolutions.push(i);
      }
    }

    console.log("Solutions with the highest score: " + bestSolutions);
  </script>
</head>
<body> </body>
</html>
```

We don't want to declare scores inside the functions that operate on scores because these are going to be different for each use of the functions. Instead, we'll pass the scores as an argument into the functions, so the functions can use any scores array to generate results.

We use this first chunk of code to output each score and at the same time compute the highest score in the array. We could put this in a printAndGetHighScore function.

And we use this second chunk of code to figure out the best results given a high score. We could put this in a getBestResults function.

Writing the printAndGetHighScore function

We've got the code for the `printAndGetHighScore` function already. It's just the code we've already written, but to make it a function we need to think through what arguments we're passing it, and if it returns anything back to us.

Now, passing in the scores array seems like a good idea because that way, we can reuse the function on other arrays with bubble scores. And we want to return the high score that we compute in the function, so the code that calls the function can do interesting things with it (and, after all, we're going to need it to figure out the best solutions).

Oh, and another thing: often you want your functions to do *one thing* well. Here we're doing two things: we're displaying all the scores in the array and we're also computing the high score. We might want to consider breaking this into two functions, but given how simple things are right now we're going to resist the temptation. If we were working in a professional environment we might reconsider and break this into two functions, `printScores` and `getHighScore`. But for now, we'll stick with one function. Let's get this code refactored:

We've created a function that expects one argument, the scores array.

```
function printAndGetHighScore(scores) {
    var highScore = 0;
    var output;
    for (var i = 0; i < scores.length; i++) {
        output = "Bubble solution #" + i + " score: " + scores[i];
        console.log(output);
        if (scores[i] > highScore) {
            highScore = scores[i];
        }
    }
    return highScore;
}
```

This code is exactly the same. Well, actually it LOOKS exactly the same, but it now uses the parameter scores rather than the global variable scores.

And we've added one line here to return the highScore to the code that called the function.

Refactoring the code using printAndGetHighScore

Now, we need to change the rest of the code to use our new function. To do so, we simply call the new function, and set the variable `highScore` to the result of the `printAndGetHighScore` function:

```
<!doctype html>
<html lang="en">
<head>
  <title>Bubble Factory Test Lab</title>
  <meta charset="utf-8">
  <script>
    var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

    function printAndGetHighScore(scores) {
        var highScore = 0;
        var output;
        for (var i = 0; i < scores.length; i++) {
            output = "Bubble solution #" + i + " score: " + scores[i];
            console.log(output);
            if (scores[i] > highScore) {
                highScore = scores[i];
            }
        }
        return highScore;
    }

    var highScore = printAndGetHighScore(scores);
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    var bestSolutions = [];

    for (var i = 0; i < scores.length; i++) {
      if (scores[i] == highScore) {
        bestSolutions.push(i);
      }
    }

    console.log("Solutions with the highest score: " + bestSolutions);
  </script>
</head>
<body> </body>
</html>
```

Here's our new function, all ready to use.

And now we just call the function, passing in the scores array. We assign the value it returns to the variable highScore.

Now we need to refactor this code into a function and make the appropriate changes to the rest of the code.

Sharpen your pencil

Let's work through this next one together. The goal is to write a function to create an array of bubble solutions that have the high score (and there might be more than one, so that's why we're using an array). We're going to pass this function the scores array and the highScore we computed with printAndGetHighScore. Finish the code below. You'll find the answer on the next page but don't peek! Do the code yourself first, so you really get it.

Here's the original code in case you need to refer to it.

```
var bestSolutions = [];
for (var i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {
        bestSolutions.push(i);
    }
}
console.log("Solutions with the highest score: " + bestSolutions);
```

We've already started this but we need your help to finish it!

```
function getBestResults(__________, ____________) {
    var bestSolutions = ______;
    for (var i = 0; i < scores.length; i++) {
        if (__________ == highScore) {
            bestSolutions.__________;
        }
    }
    return _______________;
}

var bestSolutions = _____________(scores, highScore);
console.log("Solutions with the highest score: " + bestSolutions);
```

Putting it all together...

Once you've completed refactoring your code, make all the changes to "bubbles.html", just like we have below, and reload the bubble report. You should get exactly the same results as before. But now you know your code is more organized and reusable. Create your own scores array and try some reuse!

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

    function printAndGetHighScore(scores) {
        var highScore = 0;
        var output;
        for (var i = 0; i < scores.length; i++) {
            output = "Bubble solution #" + i + " score: " + scores[i];
            console.log(output);
            if (scores[i] > highScore) {
                highScore = scores[i];
            }
        }
        return highScore;
    }

    function getBestResults(scores, highScore) {
        var bestSolutions = [];
        for (var i = 0; i < scores.length; i++) {
            if (scores[i] == highScore) {
                bestSolutions.push(i);
            }
        }
        return bestSolutions;
    }

    var highScore = printAndGetHighScore(scores);
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    var bestSolutions = getBestResults(scores, highScore);
    console.log("Solutions with the highest score: " + bestSolutions);

  </script>
</head>
<body> </body>
</html>
```

Okay, here's the new getBestResults function.

And we use the result of that function to display the best solutions in the report.

Great job! Just one more thing... can you figure out the most cost effective bubble solution? With that final bit of data, we'll definitely take over the entire bubble solution market. Here's an array with the cost of each solution you can use to figure it out.

Here's the array. Notice that it has a cost for each of the corresponding solutions in the scores array.

```
var costs = [.25, .27, .25, .25, .25, .25,
             .33, .31, .25, .29, .27, .22,
             .31, .25, .25, .33, .21, .25,
             .25, .25, .28, .25, .24, .22,
             .20, .25, .30, .25, .24, .25,
             .25, .25, .27, .25, .26, .29];
```

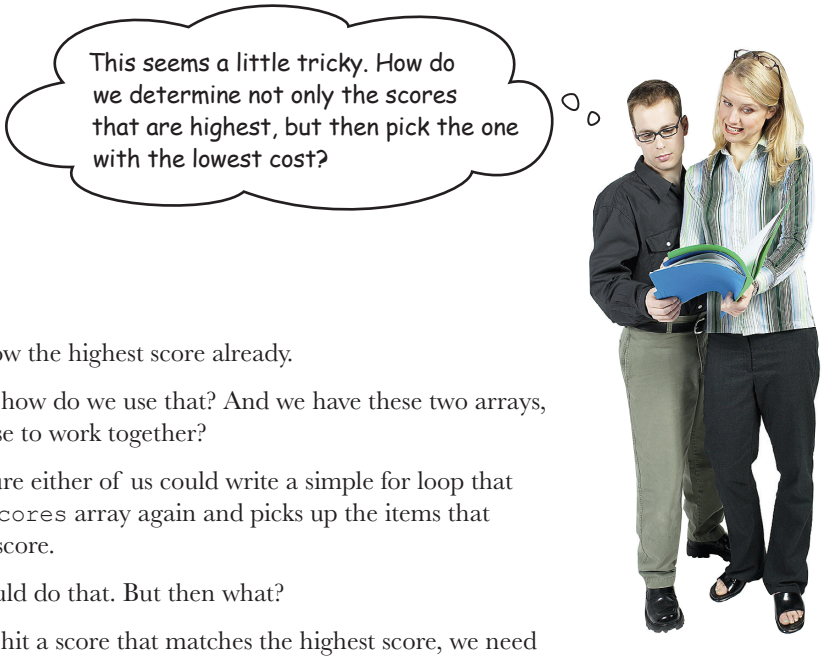
So, what's the job here? It's to take the leading bubble solutions—that is, the ones with the highest bubble scores—and choose the lowest cost one. Now, luckily, we've been given a `costs` array that mirrors the `scores` array. That is, the bubble solution score at index 0 in the `scores` array has the cost at index 0 in the `costs` array (.25), the bubble solution at index 1 in the `scores` array has a cost at index 1 in the `costs` array (.27), and so on. So, for any score you'll find its cost in the `costs` array at the same index. Sometimes we call these *parallel* arrays:

Scores and costs are parallel arrays because for each score there is a corresponding cost at the same index.

```
var costs = [.25, .27, .25, .25, .25, .25, .33, .31, .25, .29, .27, .22, ..., .29];
var scores = [60,  50,  60,  58,  54,  54,  58,  50,  52,  54,  48,  69,  ..., 44];
```

The cost at 0 is the cost of the bubble solution at 0...

And likewise for the other cost and score values in the arrays.



Judy: Well, we know the highest score already.

Frank: Right, but how do we use that? And we have these two arrays, how do we get those to work together?

Judy: I'm pretty sure either of us could write a simple for loop that goes through the `scores` array again and picks up the items that match the highest score.

Frank: Yeah, I could do that. But then what?

Judy: Anytime we hit a score that matches the highest score, we need to see if its cost is the lowest we've seen.

Frank: Oh I see, so we'll have a variable that keeps track of the index of the "lowest cost high score." Wow, that's a mouthful.

Judy: Exactly. And once we get through the entire array, whatever index is in that variable is the index of the item that not only matches the highest score, but has the lowest cost.

Frank: What if two items match in cost?

Judy: Hmm, we have to decide how to handle that. I'd say, whatever one we see first is the winner. Of course we could do something more complex, but let's stick with that unless the CEO says differently.

Frank: This is complicated enough I think I want to sketch out some pseudocode before writing anything.

Judy: I agree; whenever you are managing indices of multiple arrays things can get tricky. Let's do that; in the long run I'm sure it will be faster to plan it first.

Frank: Okay, I'll take a first stab at it…

I'm pretty sure I nailed the pseudocode. Check it out below. Now you go ahead and translate it into JavaScript. Make sure to check your answer.

```
FUNCTION GETMOSTCOSTEFFECTIVESOLUTION (SCORE, COSTS, HIGHSCORE)
DECLARE a variable cost and set to 100.
DECLARE a variable index.
FOR: var i=0; i < scores.length; i++
    IF the bubble solution at score[i] has the highest score
        IF the current value of cost is greater than the cost of the bubble solution
        THEN
            SET the value of index to the value of i
            SET the value of cost to the cost of the bubble solution
        END IF
    END IF
END FOR
RETURN index
```

```
function getMostCostEffectiveSolution(scores, costs, highscore) {
```

← Translate the pseudocode to JavaScript here.

```
}
var mostCostEffective = getMostCostEffectiveSolution(scores, costs, highScore);
console.log("Bubble Solution #" + mostCostEffective + " is the most cost effective");
```

THE WINNER: SOLUTION #11

The last bit of code you wrote really helped determine the TRUE winner; that is, the solution that produces the most bubbles at the lowest cost. Congrats on taking a lot of data and crunching it down to something Bubbles-R-Us can make real business decisions with.

Now, if you're like us, you're dying to know what is in Bubble Solution #11. Look no further; the Bubble-R-Us CEO said he'd be delighted to give you the recipe after all your unpaid work.

So, you'll find the recipe for Bubble Solution #11 below. Take some time to let your brain process arrays by making a batch, getting out, and blowing some bubbles before you begin the next chapter. Oh, but don't forget the bullet points and the crossword before you go!

Bubble Solution #11

2/3 cup dishwashing soap

1 gallon water

2 to 3 tablespoons of glycerine (available at the pharmacy or chemical supply house.)

INSTRUCTIONS: Mix ingredients together in a large bowl and have fun!

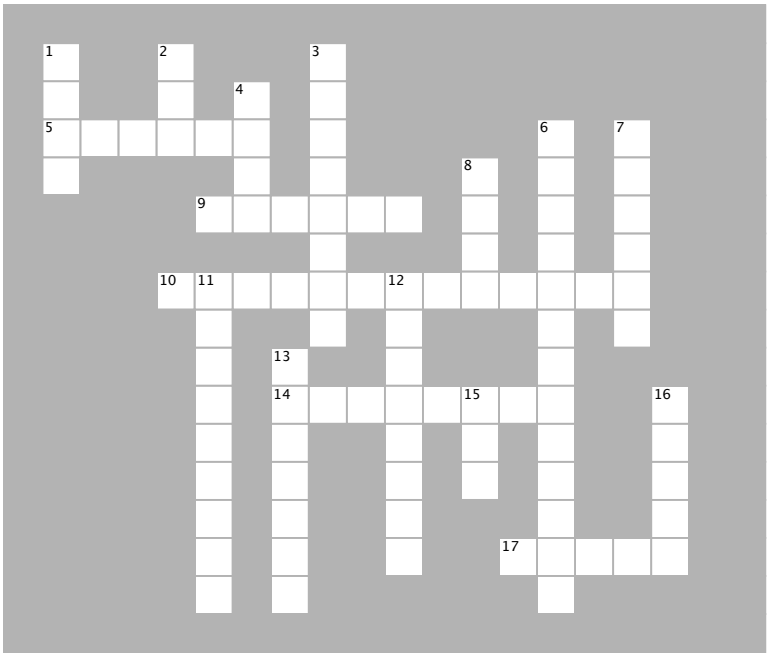
DO try this at HOME!

BULLET POINTS

- Arrays are a **data structure** for ordered data.
- An array holds a set of items, each with its own **index**.
- Arrays use a zero-based index, where the first item is at index zero.
- All arrays have a **length** property, which holds a number representing the number of items in the array.
- You can access any item using its index. For example, use myArray[1] to access item one (the second item in the array).
- If an item doesn't exist, trying to access it will result in a value of undefined.
- Assigning a value to an existing item will change its value.
- Assigning a value to an item that doesn't exist in the array will create a new item in the array.
- You can use a value of any type for an array item.
- Not all the values in an array need to be the same type.
- Use the **array literal notation** to create a new array.
- You can create an empty array with
  var myArray = [ ];
- The **for loop** is commonly used to iterate through arrays.
- A for loop packages up variable initialization, a conditional test, and variable increment into one statement.
- The while loop is most often used when you don't know how many times you need to loop, and you're looping until a condition is met. The for loop is most often used when you know the number of times the loop needs to execute.
- Sparse arrays occur when there are undefined items in the middle of an array.
- You can increment a variable by one with the **post-increment** operator ++.
- You can decrement a variable by one with the **post-decrement** operator --.
- You can add a new value to an array using **push**.

JavaScript cross

Let arrays sink into your brain as you do the crossword.



ACROSS

5. An array with undefined values is called a _____ array.

9. To change a value in an array, simply _____ the item a new value.

10. Who thought he was going to have the winning bubble solution?

14. When you _____ your code, you organize it so it's easier to read and maintain.

17. Each value in an array is stored at an _____.

DOWN

1. To add a new value to the end of an existing array, use _____.

2. We usually use a _____ loop to iterate over an array.

3. Arrays are good for storing _____ values.

4. The last index of an array is always one _____ than the length of the array.

6. The operator we use to increment a loop variable.

7. When iterating through an array, we usually use the _____ property to know when to stop.

8. The index of the first item in an array is _____.

11. The value an array item gets if you don't specify one.

12. Functions can help _____ your code.

13. An array is an _____ data structure.

15. How many bubble solutions had the highest score?

16. Access an array item using its _____ in square brackets.

Sharpen your pencil Solution

The products array below holds the Jenn and Berry's ice cream flavors. The ice creams were added to this array in the order of their creation. Finish the code to determine the most recent ice cream flavor they created. Here's our solution.

```
var products = ["Choo Choo Chocolate", "Icy Mint", "Cake Batter", "Bubblegum"];
var last = products.length - 1;
var recent = products[last];
```

We can use the length of the array, minus one to get the index of the last item. The length is 4, and the index of the last item is 3, because we start from 0.

Code Magnets Solution

We've got code for testing to see which ice cream flavors have bubblegum pieces in them. We had all the code nicely laid out on our fridge using fridge magnets, but the magnets fell on the floor. It's your job to put them back together. Be careful; a few extra magnets got mixed in. Here's our solution.

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];

var hasBubbleGum = [false,
                    false,
                    false,
                    true];

var i = 0;

while (i < hasBubbleGum.length)

{

if (hasBubbleGum[i])

{

console.log(products[i] +
    " contains bubble gum");

}

i = i + 1;

}
```

Rearrange the magnets here.

Leftover magnets.

```
{

i = i + 2;

while (i > hasBubbleGum.length)
```

Here's the output we're expecting.

JavaScript console

```
Bubblegum contains bubble gum!
```

Sharpen your pencil Solution

Rewrite your fridge magnet code (from two pages back) so that it uses a for loop instead of a while loop. If you need a hint, refer to each piece of the while loop on the previous page and see how it maps to the corresponding location in the for loop. Here's our solution.

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
var hasBubbleGum = [false,
                    false,
                    false,
                    true];
var i = 0;
while (i < hasBubbleGum.length) {
    if (hasBubbleGum[i]) {
        console.log(products[i] +
            " contains bubble gum");
    }
    i = i + 1;
}
```

Your code goes here.

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
var hasBubbleGum = [false,
                    false,
                    false,
                    true];
for (var i = 0; i < hasBubbleGum.length; i = i + 1) {
    if (hasBubbleGum[i]) {
        console.log(products[i] + " contains bubble gum");
    }
}
```

Sharpen your pencil Solution

Go ahead and implement the pseudocode on the previous page to find the highest score by filling in the blanks in the code below. Once you're done, give it a try in the browser by updating the code in "bubbles.html", and reloading the page. Check the results in the console, and fill in the blanks in our console display below with the number of bubble tests and the highest score. Here's our solution.

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

var highScore = 0;
var output;
for (var i = 0; i < scores.length; i++) {
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (scores[i] > highScore) {
        highScore = scores[i];
    }
}
console.log("Bubbles tests: " + scores.length);
console.log("Highest bubble score: " + highScore);
```

Fill in the blanks to complete the code here...

... and then fill in the blanks showing the output you get in the console.

JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: 36
Highest bubble score: 69
```

Here's our solution for the getMostCostEffectiveSolution function, which takes an array of scores, an array of costs, and a high score, and finds the index of the bubble solution with the highest score and lowest cost. Go ahead and test drive all your code in "bubbles.html" and make sure you see the same results.

The getMostCostEffectiveSolution takes the array of scores, the array of costs, and the high score.

```
function getMostCostEffectiveSolution(scores, costs, highscore) {
    var cost = 100;
    var index;

    for (var i = 0; i < scores.length; i++) {
        if (scores[i] == highscore) {
            if (cost > costs[i]) {
                index = i;
                cost = costs[i];
            }
        }
    }
    return index;
}
var mostCostEffective = getMostCostEffectiveSolution(scores, costs, highScore);
console.log("Bubble Solution #" + mostCostEffective + " is the most cost effective");
```

We'll keep track of the lowest cost solution in cost...

We start cost at a high number, and we'll lower it each time we find a lower cost solution (with a high score).

... and the index of the lowest cost solution in index.

We iterate through the scores array like before...

... and check to see if the score has the high score.

If it does, then we can check its cost. If the current cost is greater than the solution's cost, then we've found a lower cost solution, so we'll make sure we keep track of which solution it is (its index in the array) and store its cost in the cost variable as the lowest cost we've seen so far.

Once the loop is complete, the index of solution with the lowest cost is stored in index, so we return that to the code that called the function.

And then display the index (which is the bubble solution #) in the console.

JavaScript console

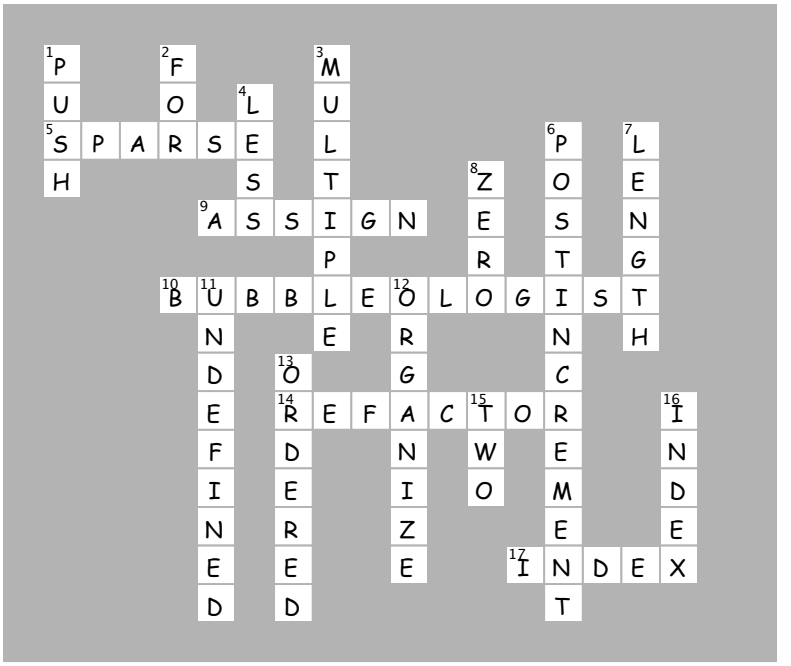
```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11,18
Bubble Solution #11 is the most cost effective
```

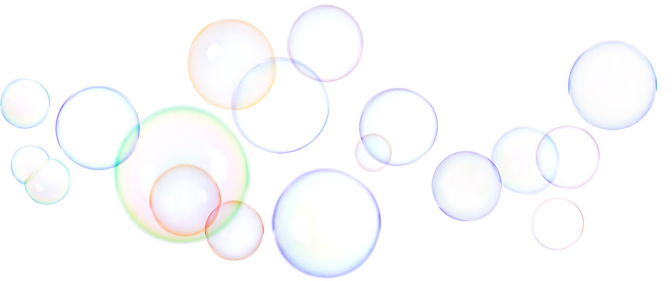
The final report showing bubble solution #11 as the winner of the bubble tests for having the highest bubble factor at a low cost.

BONUS: We could also implement this using the bestSolutions array so we wouldn't have to iterate through all the scores again. Remember, the bestSolutions array has the indices of the solutions with the highest scores. So in that code, we'd use the items in the bestSolutions array to index into the costs array to compare the costs. The code is a little more efficient than this version, but it's also a little bit more difficult to read and understand! If you're interested, we've included the code in the book code download at wickedlysmart.com.

JavaScript cross Solution

Let arrays sink into your brain as you do the crossword.





5 understanding objects

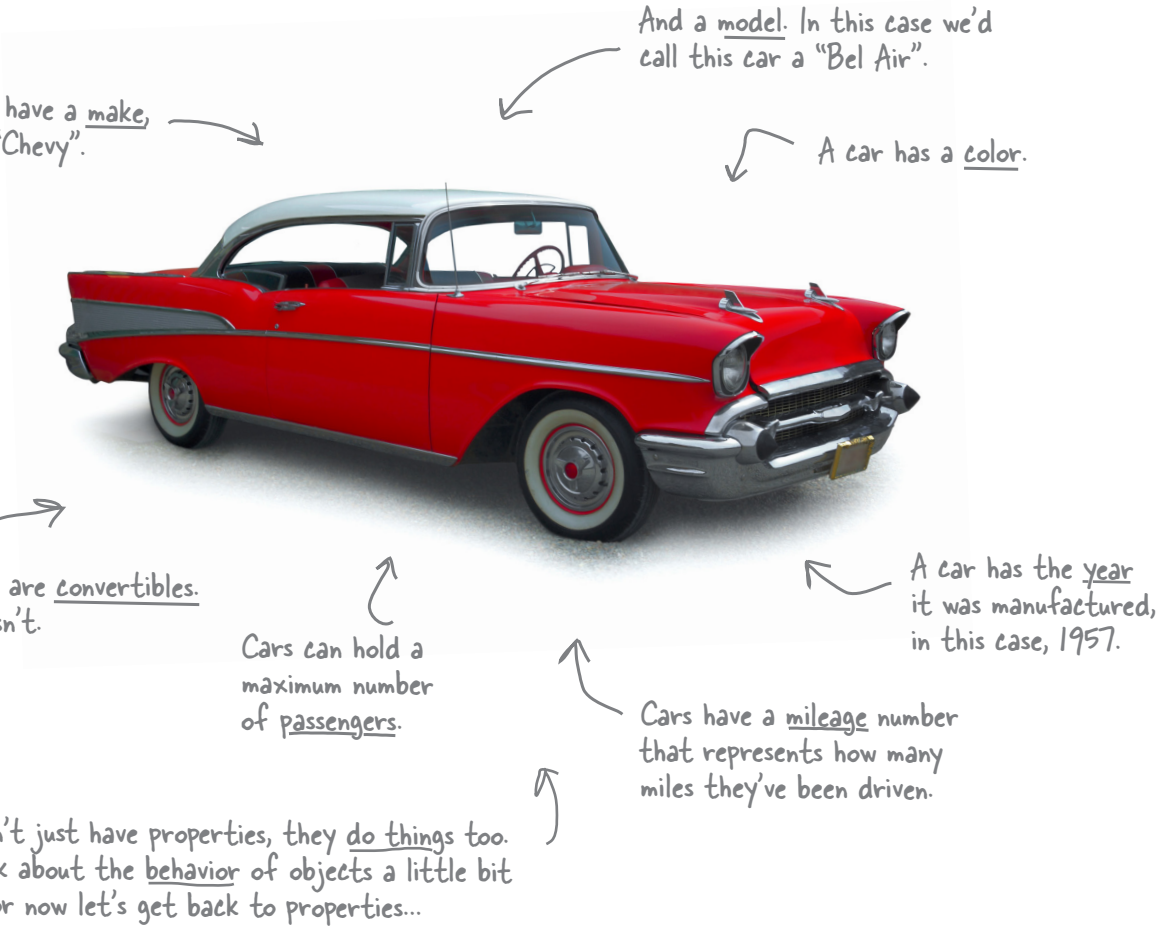
A trip to Objectville

So far you've been using primitives and arrays in your code. And, you've approached coding in quite a **procedural manner** using simple statements, conditionals and for/while loops with functions—that's not exactly **object-oriented**. In fact, it's not object-oriented *at all!* We did use a few objects here and there without really knowing it, but you haven't written any of your own objects yet. Well, the time has come to leave this boring procedural town behind to create some **objects** of your own. In this chapter, you're going to find out why using objects is going to make your life so much better—well, better in a **programming sense** (we can't really help you with your fashion sense *and* your JavaScript skills all in one book). Just a warning: once you've discovered objects you'll never want to come back. Send us a postcard when you get there.

Did someone say "Objects"?!

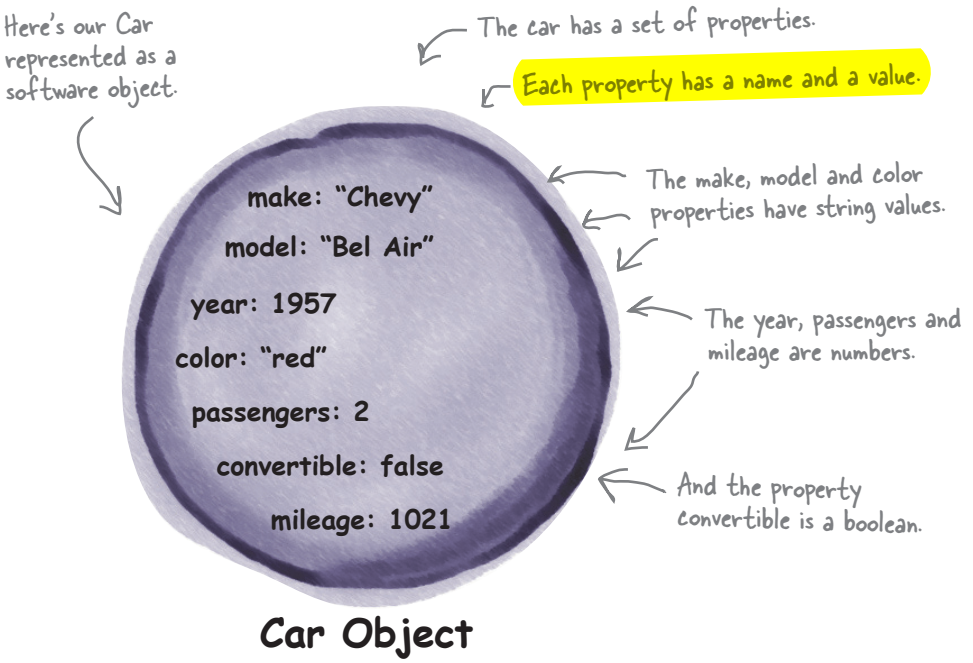
Ah, our favorite topic! Objects are going to take your JavaScript programming skills to the next level—they're the key to managing complex code, to understanding the browser's document model (which we'll do in the next chapter), to organizing your data, and they're even the fundamental way many JavaScript libraries are packaged up (more on that much later in the book). That said, objects are a difficult topic, right? Hah! We're going to jump in head first and you'll be using them in no time.

Here's the secret to JavaScript objects: they're just a collection of properties. Let's take an example, say, a car. A car's got properties:



Thinking about properties...

Of course there's a lot more to a real car than just a few properties, but for the purposes of coding, these are the properties we want to capture in software. Let's think about these properties in terms of JavaScript data types:



Are there other properties you'd want to have in a car object? Go ahead and think through all the properties you might come up with for a car and write them below. Remember, only some real-world properties are going to be useful in software.

Those fuzzy dice may look nice, but would they really be useful in an object?

Sharpen your pencil

We've started making a table of property names and values for a car. Can you help complete it? Make sure you compare your answers with ours before moving on!

Put your property names here.

And put the corresponding values over here.

```
{
  make       : "Chevy",
  model      :
  year       :
  color      :
  passengers :
  convertible:
  mileage    :
             :
             :
};
```

Put your answers here. Feel free to expand the list to include your own properties.

When you're done notice the syntax we've placed around the properties and values. There might be a pop-quiz at some point... just sayin'.

BRAIN POWER

What if the car is a taxi? What properties and values would it share with your '57 Chevy? How might they differ? What additional properties might it have (or not have)?

How to create an object

Here's the good news: after the last **Sharpen your Pencil** exercise, you're already most of the way to creating an object. All you really need to do is assign what you wrote on the previous page to a variable (so you can do things with your object after you've created it). Like this:

Add a variable declaration for the object.

Next, start an object with a left curly brace.

Then all the object's properties go inside.

Each property has a name, a colon and then a value. Here we have strings, numbers and one boolean as property values.

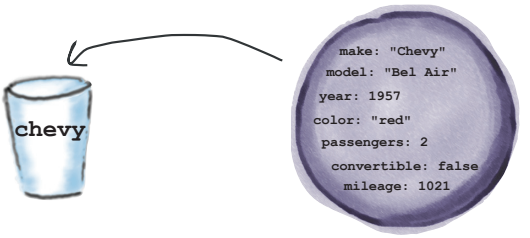
Notice that each property is separated by a comma.

We end the object with a closing curly brace, and just like any other variable declaration, we end this one with a semicolon.

```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};
```

The result of all this? A brand new object of course. Think of the object as something that holds all your names and values (in other words, your properties) together.

Now you've got a live object complete with a set of properties. And you've assigned your object to a variable that you can use to access and change its properties.



We've taken the textual description of the object above and created a real live JavaScript object from it.

You can now take your object, pass it around, get values from it, change it, add properties to it, or take them away. We'll get to how to do all that in a second. For now, let's create some more objects to play with…

You don't have to be stuck with just one object. The real power of objects (as you'll see soon enough) is having lots of objects and writing code that can operate on whatever object you give it. Try your hand at creating another object from scratch… another car object. Go ahead and work out the code for your second object.

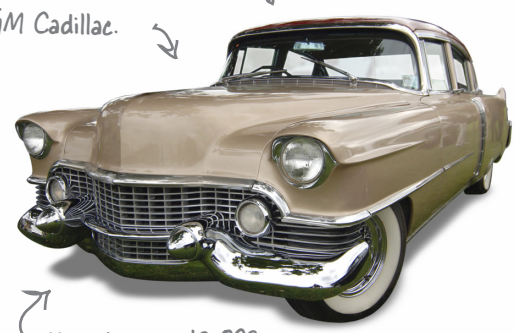
```
var cadi = {

};
```

Put the properties for your Cadillac object here.

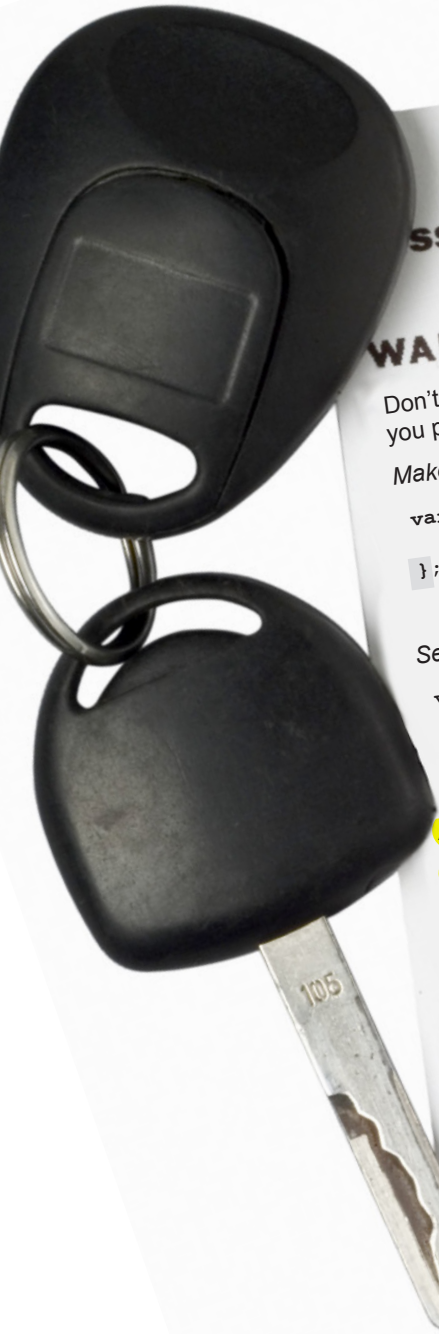
We'll call this a tan color.

This is a 1955 GM Cadillac.

It's not a convertible, and it can hold five passengers (it's got a nice big bucket seat in the back).



Its mileage is 12,892.



SPEEDING TICKET

Issued by the Webville Police dept.

WARNING CITATION No. 10

Don't worry, you're getting off easy this time; rather than issuing a ticket, we ask that you please review the following "rules of the road" for creating objects.

Make sure you enclose your object in curly braces:

```
var cat = {
  name: "fluffy"
};
```

Separate the property name and property value with a colon:

```
var planet = {
  diameter: 49528
};
```

A property name can be any string, but we usually stick with valid variable names:

```
var widget = {
  cost$: 3.14,
  "on sale": true
};
```

Notice that if you use a string with a space in it for a property name, you need to use quotes around the name.

No two properties in an object can have the same name:

```
var forecast = {
  highTemp: 82,
  highTemp: 56
};
```

WRONG! This won't work.

Separate each property name and value pair with a comma:

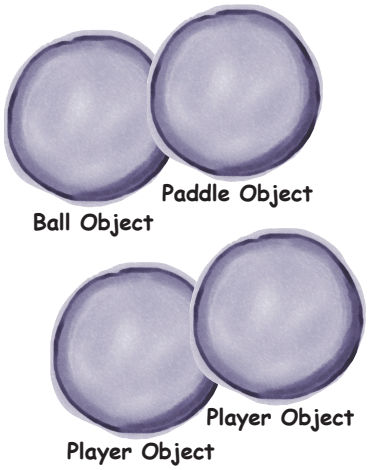
```
var gadget = {
  name: "anvil",
  isHeavy: true
};
```

Don't use a comma after the last property value:

```
var superhero = {
  name: "Batman",
  alias: "Caped Crusader"
};
```

No comma needed here!

What is Object-Oriented Anyway?



Up 'til now, we've been thinking of a problem as a set of variable declarations, conditionals, for/while statements, and function calls. That's thinking *procedurally*: first do this, then do this and so on. With *object-oriented* programming we think about a problem in terms of objects. Objects that have state (like a car might have an oil and a fuel level), and behavior (like a car can be started, driven, parked and stopped).

What's the point? Well, object-oriented programming allows you to free your mind to think at a higher level. It's the difference between having to toast your bread from first principles (create a heating coil out of wire, hook it to electricity, turn the electricity on and then hold your bread close enough to toast it, not to mention watch long enough for it to toast and then unhook the heating coil), and just using a toaster (place bread in toaster and push down on the toast button). The first way is procedural, while the second way is object-oriented: you have a toaster object that supports an easy method of inserting bread and toasting it.

What do you like about OO?

"It helps me design in a more natural way. Things have a way of evolving."
-Joy, 27, software architect

"Not messing around with code I've already tested, just to add a new feature."
-Brad, 32, programmer

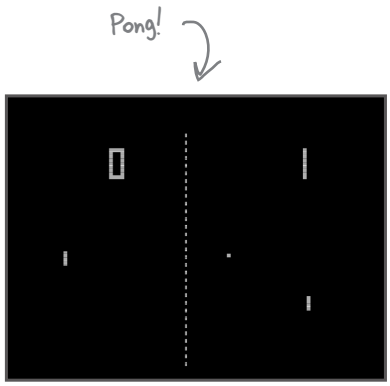
"I like that the data and the methods that operate on that data are together in one object."
-Josh, 22, beer drinker

"Reusing code in other apps. When I write a new object, I can make it flexible enough to be used in something new, later."
-Chris, 39, project manager

"I can't believe Chris just said that. He hasn't written a line of code in five years."
-Daryl, 44, works for Chris

BRAIN POWER

Say you were implementing a classic ping-pong style video arcade game. What would you choose as objects? What state and behavior do you think they'd have?

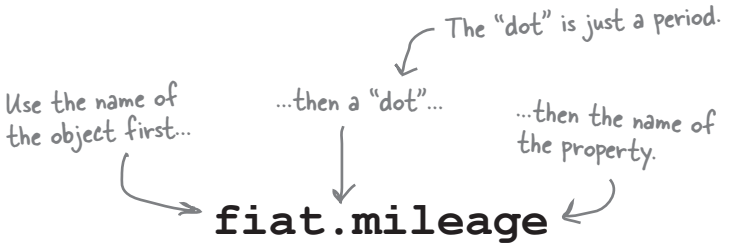



```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};
```

How properties work

So you've got all your properties packaged up in an object. Now what? Well, you can examine the values of those properties, change them, add new properties, take away properties, and in general, compute using them. Let's try a few of these things out, using JavaScript of course.

How to access a property. To access a property in an object, start with the object name, follow it with a period (otherwise known as a "dot") and then use the property name. We often call that "dot" notation and it looks like this:



And then we can use a property in any expression, like this:

```
var miles = fiat.mileage;
if (miles < 2000) {
    buyIt();
}
```

Start with the variable that holds your object, add a period (otherwise known as a dot) and then your property name.

Dot Notation .

- Dot notation (.) gives you access to an object's properties.
- For example, `fiat.color` is a property in `fiat` with the name `color` and the value "Medium Blue".

How to change a property. You can change the value of a property at any time. All you need to do is assign the property to a new value. Like, let's say we wanted to set the mileage of our nifty Fiat to an even 10,000. You'd do it like this:

```
fiat.mileage = 10000;
```

Just specify the property you want to change and then give it a new value. Note: in some states this may be illegal!

How to add a new property. You can extend your object at any time with new properties. To do this you just specify the new property and give it a value. For instance, let's say we want to add a boolean that indicates when the Fiat needs to be washed:

```
fiat.needsWashing = true;
```

As long as the property doesn't already exist in the object, it's added to the object. Otherwise, the property with this name is updated.

The new property is added to your object.

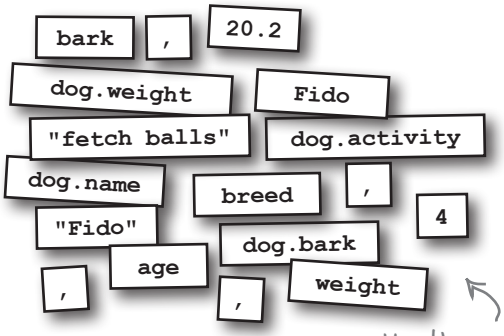
How to compute with properties. Computing with properties is simple: just use a property like you would any variable (or any value). Here are a few examples:

You can use an object's property just like you use a variable, except you need to use dot notation to access the property in the object.

```
if (fiat.year < 1965) {
    classic = true;
}
for (var i = 0; i < fiat.passengers; i++) {
    addPersonToCar();
}
```

Object Magnets

This code got all scrambled up on the fridge. Practice your object creating and dot notation skills by getting it all back together. Be careful, some extra magnets might have got mixed in!



Use these magnets to complete the code.

The dog object.

```
var dog = {
    name: _________
    _________: 20.2
    age: ________
    ________: "mixed",
    activity: ____________
};
var bark;
if (_____________ > 20) {
   bark = "WOOF WOOF";
} else {
   bark = "woof woof";
}
var speak = __________ + " says " + ________ + " when he wants to " + _____________;
console.log(speak);
```

Fido is hoping you get all his properties right.

Yes, you can add or delete properties at any time. As you know, to add a property to an object you simply assign a value to a new property, like this:

```
fido.dogYears = 35;
```

and from that point on `fido` will have a new property `dogYears`. Easy enough.

To delete a property, we use a special keyword, wait for it... `delete`. You use the `delete` keyword like this:

```
delete fido.dogYears;
```

When you delete a property, you're not just deleting the value of the property, you're deleting the property itself. And, if you try to use `fido.dogYears` after deleting it, it will evaluate to `undefined`.

The `delete` expression returns `true` if the property was deleted successfully. `delete` will return false only if it can't delete a property (which could happen for, say, a protected object that belongs to the browser). It will return `true` even if the property you're trying to delete doesn't exist in the object.

there are no Dumb Questions

Q: How many properties can an object have?

A: As few or as many as you want. You can have an object with no properties, or you can have an object with hundreds of properties. It's really up to you.

Q: How can I create an object with no properties?

A: Just like you create any object, only leave out all the properties. Like this:

```
var lookMaNoProps = { };
```

Q: I know I just asked how to create an object with no properties, but why would I want to do this?

A: Well, you might want to start with an entirely empty object and then add your own properties dynamically, depending on the logic of your code. This way of creating an object will become clear as we continue to use objects.

```
var lookMaNoProps = { };
lookMaNoProps.age = 10;
if (lookMaNoProps.age > 5) {
  lookMaNoProps.school = "Elementary";
}
```

Q: What's better about an object than just using a bunch of variables? After all, each of the properties in the fiat object could just be its own variable, right?

A: Objects package up the complexity of your data so that you can focus on the high level design of your code, not the nitty gritty details. Say you want to write a traffic simulator with tens of cars; you'll want to focus on cars and streetlights and road objects and not hundreds of little variables. Objects also make your life easier because they encapsulate, or hide, the complexity of the state and behavior of your objects so you don't have to worry about them. How all that works will become much clearer as you gain experience with objects.

Q: If I try to add a new property to my object, and the object already has a property with that name, what happens?

A: If you try to add a new property, like needsWashing, to fiat and fiat already has a property needsWashing, then you'll be changing the existing value of the property. So if you say:

```
fiat.needsWashing = true;
```

but fiat already contains a property needsWashing with the value false, then you're changing the value to true.

Q: What happens if I try to access a property that doesn't exist? Like if I said,

```
if (fiat.make) { ... }
```

but fiat didn't have a property make?

A: The result of the expression fiat.make will be undefined if fiat doesn't have a property named make.

Q: What happens if I put a comma after the last property?

A: In most browsers it won't cause an error. However, in older versions of some browsers this will cause your JavaScript to halt execution. So, if you want your code to work in as many browsers as possible, keep away from extraneous commas.

Q: Can I use console.log to display an object in the console?

A: You can. Just write:

```
console.log(fiat);
```

in your code, and when you load the page with the console open, you'll see information about the object displayed in the console.

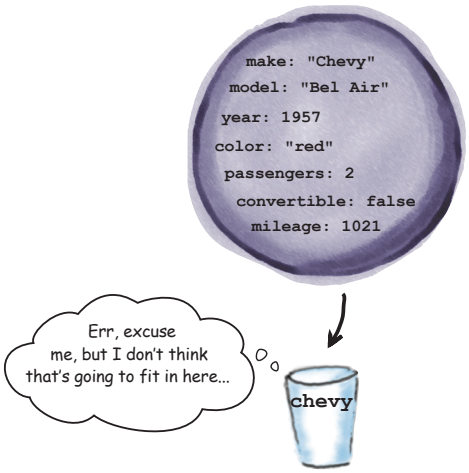
JavaScript console

```
> console.log(fiat)
Object {make: "Fiat", model: "500", year: 1957,
color: "Medium Blue", passengers: 2…}
>
```

How does a variable hold an object? Inquiring minds want to know...

You've already seen that a variable is like a container and it holds a value. But numbers, strings and booleans are pretty small values. What about objects? Can a variable hold any sized object no matter how many properties you put in it?

- Variables don't actually hold objects.
- Instead they hold a reference to an object.
- The reference is like a pointer or an address to the actual object.
- In other words, a variable doesn't hold the object itself, but it holds something like a pointer. And, in JavaScript we don't really know *what* is inside a reference variable. We *do* know that whatever it is, it points to our object.
- When we use dot notation, the JavaScript interpreter takes care of using the reference to get the object and then accesses its properties for us.



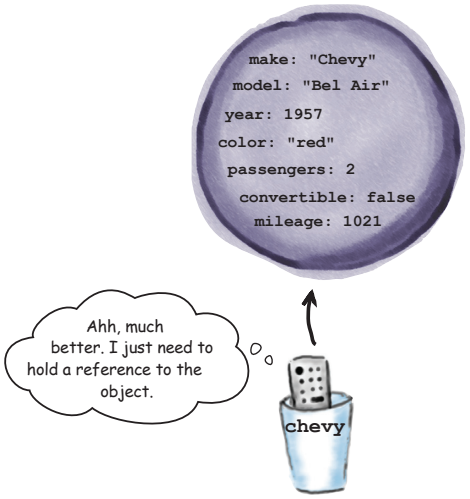
So, you can't stuff an object into a variable, but we often think of it that way. It's not what happens though—there aren't giant expandable cups that can grow to the size of any object. Instead, an object variable just holds a reference to the object.

Here's another way to look at it: a primitive variable represents the actual *value* of the variable while an object variable represents *a way to get to the object*. In practice you'll only need to think of objects as, well, objects, like dogs and cars, not as references, but knowing variables contain *references* to objects will come in handy later (and we'll see that in just a few pages).

And also think about this: you use the dot notation (.) on a reference variable to say, "use the reference *before* the dot to get me the object that has the property *after* the dot." (Read that sentence a few times and let it sink in.) For example:

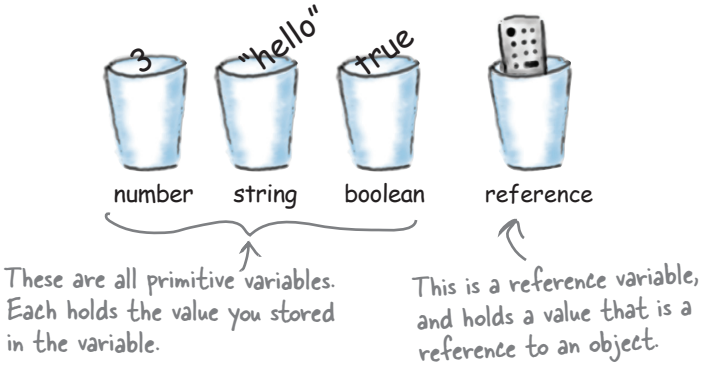
```
car.color;
```

means "use the object referenced by the variable `car` to access the `color` property."



Comparing primitives and objects

Think of an object reference as just another variable value, which means that we can put that reference in a cup, just like we can primitive values. With primitive values, the value of a variable is… the *value*, like 5, -26.7, "hi", or false. With reference variables, the value of the variable is a *reference*: a value that represents a way to get to a specific object.



Initializing a primitive variable

When you declare and initialize a primitive, you give it a value, and that value goes right in the cup, like this:

```
var x = 3;
```

The variable holds the number three.

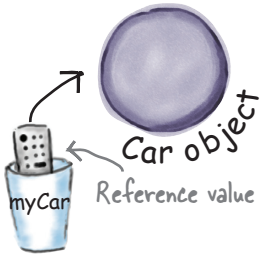
Initializing an object (a reference) variable

When you declare and initialize an object, you make the object using object notation, but that object won't fit in the cup. So what goes in the cup is a *reference* to the object.

```
var myCar = {...};
```

A reference to the Car object goes into the variable.

The Car object itself does not go into the variable!



We don't know (or care) how the JavaScript interpreter represents object references.

We just know we can access an object and its properties using dot notation.

Doing even more with objects...

Let's say you're looking for a good car for your stay in Webville. Your criteria? How about:

- ❏ Built in 1960 or before.
- ❏ 10,000 miles or less.

You also want to put your new coding skills to work (and make your life easier) so you want to write a function that will "prequalify" cars for you—that is, if the car meets your criteria then the function returns true; otherwise the car isn't worth your time and the function returns false.

More specifically, you're going to write a *function* that *takes a car object as a parameter* and puts that car through the test, returning a boolean value. Your function is going to work for *any car object*.

Let's give it a shot:

Here's the function.

You're going to pass it a car object.

```
function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}
```

Just use dot notation on the car parameter to access the mileage and year properties.

Test each property value against the prequalification criteria.

If either of the disqualification tests succeeds we return false. Otherwise we return true, meaning we've successfully prequalified!

Now let's give this function a try. First you need a car object. How about this one:

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

What do you think? Should we consider this yellow taxi? Why or why not?

Doing some pre-qualification

We've done enough talking about objects. Let's actually create one and put it through its paces using the `prequal` function. Grab your favorite, basic HTML page ("prequal.html") and throw in the code below, load the page and see if the taxi qualifies:

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};

function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}

var worthALook = prequal(taxi);

if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```

Does the taxi cut it?

Here's what we got... let's quickly trace through the code on the next page to see how the Taxi got rejected...

JavaScript console

```
You should really pass on the Webville Motors Taxi
```

Stepping through pre-qualification

1. First we create the taxi object and assign it to the variable `taxi`. Of course, the `taxi` variable holds a reference to the taxi object, not the object itself.

```
var taxi = { ... };
```

2. Next we call `prequal`, passing it the argument `taxi`, which is bound to the parameter `car` in the function.

```
function prequal(car) {
    ...
}
```

car points to the same object as taxi!

3. We then perform the tests in the body of the function, using the taxi object in the car parameter.

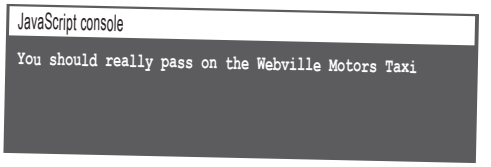
```
if (car.mileage > 10000) {
    return false;
} else if (car.year > 1960) {
    return false;
}
```

In this case, the taxi's mileage is way above 10,000 miles, so prequal returns false. Too bad; it's a cool ride.

4. Unfortunately the taxi has a lot of miles, so the first test of car.mileage > 10000 is true. The function returns false, and so `worthALook` is set to false. We then get "You should really pass on the Webville Motors Taxi" displayed in the console.

The prequal function returns false, and so we get...

```
var worthALook = prequal(taxi);

if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```



Sharpen your pencil

Your turn. Here are three more car objects; what is the result of passing each car to the `prequal` function? Work the answer by hand, and then write the code to check your answers:

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};

prequal(cadi);
```

_____

Write the value of prequal here.

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

_____

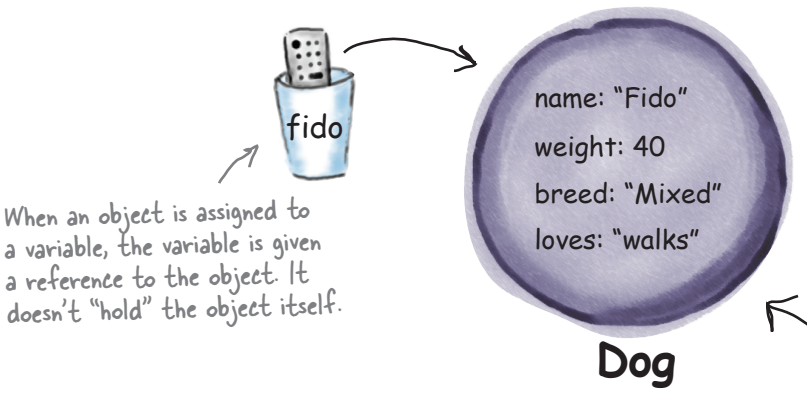
```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};

prequal(chevy);
```

_____

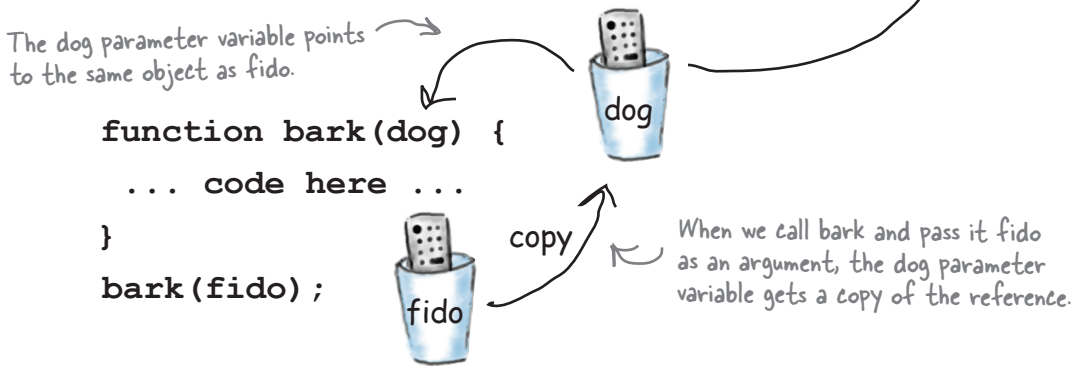
Let's talk a little more about passing objects to functions

We've already talked a bit about how arguments are passed to functions—arguments are *passed by value*, which means *pass-by-copy*. So if we pass an integer, the corresponding function parameter gets a copy of the value of that integer for its use in the function. The same rules hold true for objects, however, we should look a little more closely at what pass-by-value means for objects to understand what happens when you pass an object to a function.

You already know that when an object is assigned to a variable, that variable holds a *reference* to the object, not the object itself. Again, think of a reference as a pointer to the object:



So, when you call a function and pass it an object, you're passing the *object reference*, not the object itself. So using our pass by value semantics, a copy of the reference is passed into the parameter, and that reference remains a pointer to the original object.



So, what does this all mean? Well, one of the biggest ramifications is that if you change a property of the object in a function, you're changing the property in the *original* object. So any changes you make to the object inside a function will still be there when the function completes. Let's step through an example...

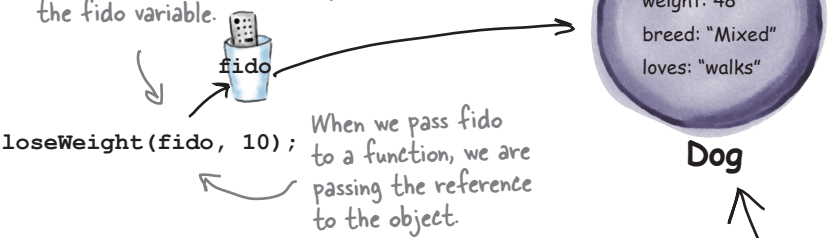
Putting Fido on a diet....

Let's say we are testing a new method of weight loss for dogs, which we want to neatly implement in a function `loseWeight`. All you need to do is pass `loseWeight` your dog object and an amount to lose, and like magic, the dog's weight will be reduced. Here's how it works:

1. First check out the dog object, fido, which we are going to pass to the loseWeight function:

Here's the dog object.

fido is a reference to an object, which means the object doesn't live in the fido variable, but is pointed to by the fido variable.




```
loseWeight(fido, 10);
```

When we pass fido to a function, we are passing the reference to the object.

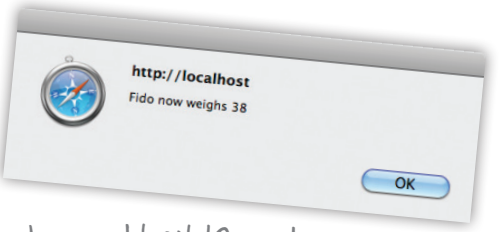
2. The dog parameter of the loseWeight function gets a copy of the reference to fido. So any changes to the properties of the parameter variable affect the object that was passed in.

When we pass fido into loseWeight, what gets assigned to the dog parameter is a copy of the reference, not a copy of the object. So fido and dog point to the same object.

The dog reference is a copy of the fido reference.


```
function loseWeight(dog, amount) {
  dog.weight = dog.weight - amount;
}

alert(fido.name + " now weighs " + fido.weight);
```

So, when we subtract 10 pounds from dog.weight, we're changing the value of fido.weight.



Sharpen your pencil

You've been given a super secret file and two functions that allow access to get and set the contents of the file, but only if you have the right password. The first function, `getSecret`, returns the contents of the file if the password is correct, and logs each attempt to access the file. The second function, `setSecret`, updates the contents of the file, and resets the access tracking to 0. It's your job to fill in the blanks below to complete the JavaScript and test your functions.

```
function getSecret(file, secretPassword) {
    _______.opened = _______.opened + 1;
    if (secretPassword == _______.password) {
        return ______.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == _______.password) {
        ______.opened = 0;
        ______.contents = secret;
    }
}

var superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
var secret = getSecret(_________________, _____);
console.log(secret);

setSecret(_________________, _____, "Dr. Evel's next meeting is in Philadelphia.");
secret = getSecret(_________________, _____);
console.log(secret);
```

I'm back, and this time I've got an Auto-O-Matic. This baby will have you hawking new cars all day long.

```
<!doctype html>
<html lang="en">
<head>
  <title>Object-o-matic</title>
  <meta charset="utf-8">
  <script>
    function makeCar() {
        var makes = ["Chevy", "GM", "Fiat", "Webville Motors", "Tucker"];
        var models = ["Cadillac", "500", "Bel-Air", "Taxi", "Torpedo"];
        var years = [1955, 1957, 1948, 1954, 1961];
        var colors = ["red", "blue", "tan", "yellow", "white"];
        var convertible = [true, false];

        var rand1 = Math.floor(Math.random() * makes.length);
        var rand2 = Math.floor(Math.random() * models.length);
        var rand3 = Math.floor(Math.random() * years.length);
        var rand4 = Math.floor(Math.random() * colors.length);
        var rand5 = Math.floor(Math.random() * 5) + 1;
        var rand6 = Math.floor(Math.random() * 2);

        var car = {
            make: makes[rand1],
            model: models[rand2],
            year: years[rand3],
            color: colors[rand4],
            passengers: rand5,
            convertible: convertible[rand6],
            mileage: 0
        };
        return car;
    }

    function displayCar(car) {
        console.log("Your new car is a " + car.year + " " + car.make + " " + car.model);
    }

    var carToSell = makeCar();
    displayCar(carToSell);

  </script>
</head>
<body></body>
</html>
```

The Auto-O-Matic is similar to the Phrase-O-Matic from Chapter 4, except that the words are car properties, and we're generating a new car object instead of a marketing phrase!

Check out what it does and how it works.

The Auto-O-Matic

Brought to you by the same guy who brought you the Phrase-O-Matic, the Auto-O-matic creates knock-off cars all day long. That is, instead of generating marketing messages, this code generates makes, models, years and all the properties of a car object. *It's your very own car factory in code*. Let's take a closer look at how it works.

1. First, we have a `makeCar` function that we can call whenever we want to make a new car. We've got four arrays with the makes, models, years and colors of cars, and an array with true and false options for whether a car is a convertible. We generate five random numbers so we can pick a make, a model, a year, a color, and whether a car is a convertible randomly from these five arrays. And we generate one more random number we're using for the number of passengers.

We have several makes, models, years and colors to choose from in these four arrays...

```
var makes = ["Chevy", "GM", "Fiat", "Webville Motors", "Tucker"];
var models = ["Cadillac", "500", "Bel-Air", "Taxi", "Torpedo"];
var years = [1955, 1957, 1948, 1954, 1961];
var colors = ["red", "blue", "tan", "yellow", "white"];
var convertible = [true, false];
```

... and we'll use this array to choose a convertible property value, either true or false.

We're going to combine values from the arrays randomly using these four random numbers.

```
var rand1 = Math.floor(Math.random() * makes.length);
var rand2 = Math.floor(Math.random() * models.length);
var rand3 = Math.floor(Math.random() * years.length);
var rand4 = Math.floor(Math.random() * colors.length);
var rand5 = Math.floor(Math.random() * 5) + 1;
var rand6 = Math.floor(Math.random() * 2);
```

We'll use this random number for the number of passengers. We're adding 1 to the random number so we can have at least one passenger in the car.

... and we'll use this random number to choose whether a car is convertible or not.

2. Instead of creating a string by mixing and matching the various car properties, like we did with Phrase-O-Matic, this time we're creating a new object, `car`. This car has all the properties you'd expect. We pick values for the `make`, `model`, `year` and `color` properties from the arrays using the random numbers we created in step 1, and also add the `passengers`, `convertible` and `mileage` properties:

```
var car = {
    make: makes[rand1],
    model: models[rand2],
    year: years[rand3],
    color: colors[rand4],
    passengers: rand5,
    convertible: convertible[rand6],
    mileage: 0
};
```

We're creating a new car object, with property values made from the values in the arrays.

We're also setting the number of passengers to the random number we created, and setting the convertible property to true or false using the convertible array.

Finally, we're just setting the mileage property to 0 (it is a new car, after all).

3 The last statement in `makeCar` returns the new `car` object:

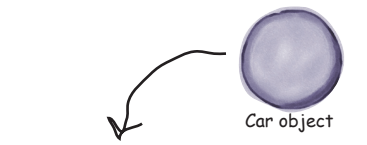
```
return car;
```

Returning an object from a function is just like returning any other value. Let's now look at the code that calls `makeCar`:

```
function displayCar(car) {
  console.log("Your new car is a " + car.year + " " +
              car.make + " " + car.model);
}
var carToSell = makeCar();
displayCar(carToSell);
```

First we call the `makeCar` function and assign the value it returns to `carToSell`. We then pass the car object returned from `makeCar` to the function `displayCar`, which simply displays a few of its properties in the console.

Don't forget; what you're returning (and assigning to the carToSell variable) is a reference to a car object.



4 Go ahead and load up the Auto-O-Matic in your browser ("autoomatic.html") and give it a whirl. You'll find no shortage of new cars to generate, and remember there's a sucker born every minute.

Here's your new car! We think a '57 Fiat Taxi would be a cool car to have.

JavaScript console

```
Your new car is a 1957 Fiat Taxi
Your new car is a 1961 Tucker 500
Your new car is a 1948 GM Torpedo
```

Reload the page a few times like we did!

Oh Behave! Or, how to add behavior to your objects

You didn't think objects were just for storing numbers and strings did you? Objects are *active*. Objects can *do things*. Dogs don't just sit there... they bark, run, and play catch, and a dog object should too! Likewise, we drive cars, park them, put them in reverse and make them brake. Given everything you've learned in this chapter, you're all set to add behavior to your objects. Here's how we do that:

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    drive: function() {
      alert.log("Zoom zoom!");
    }
};
```

You can add a function directly to an object like this.

All you do is assign a function definition to a property. Yup, properties can be functions too!

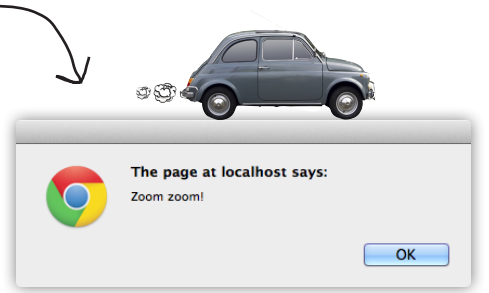
Notice we don't supply a name in the function definition, we just use the function keyword followed by the body. The name of the function is the name of the property.

And a bit of nomenclature: we typically refer to functions inside an object as methods. That is a common object-oriented term for a function in an object.

To call the `drive` function—excuse us—to call the drive *method*, you use dot notation again, this time with the object name `fiat` and the property name `drive`, only we follow the property name with parentheses (just like you would when you call any other function).

```
fiat.drive();
```

We use the dot notation to access the function in fiat, just like we would any other property. We say we're "calling the drive method in the fiat object".



The result of calling the fiat's drive method.

Improving the drive method

Let's make the fiat a little more car-like in behavior. Most cars can't be driven until the engine is started, right? How about we model that behavior? We'll need the following:

- ❏ **A boolean property to hold the state of the car (the engine is either on or off).**
- ❏ **A couple of methods to start and stop the car.**
- ❏ **A conditional check in the drive method to make sure the car is started before we drive it.**

We'll begin by adding a boolean `started` property along with methods to `start` and `stop` the car, then we'll update the `drive` method to use the `started` property.

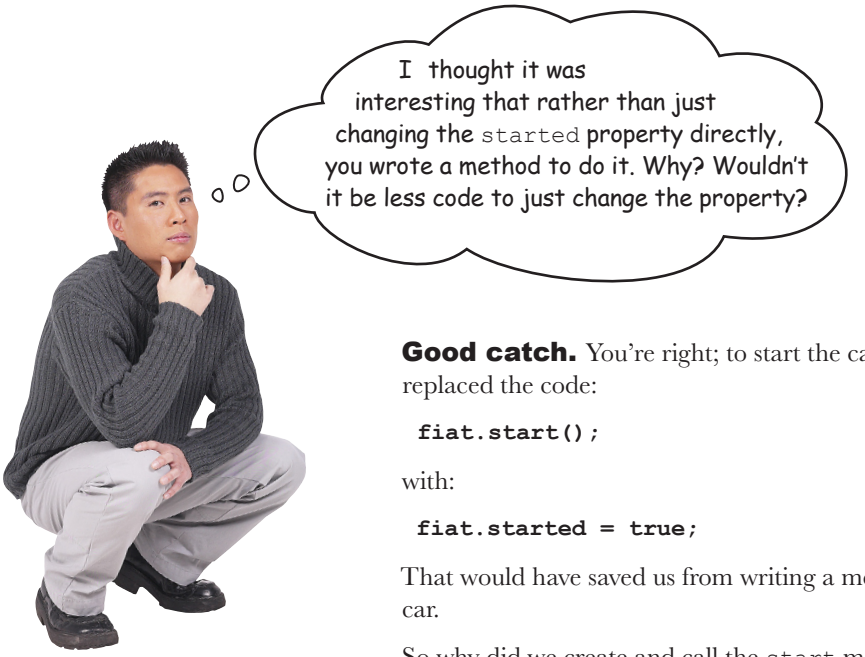
```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,

    start: function() {
       started = true;
    },

    stop: function() {
        started = false;
    },

    drive: function() {
        if (started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Here's the property to hold the current state of the engine (true if it is started and false if it is off).

And here's a method to start the car. All it does (for now) is set the started property to true.

And here's a method to stop the car. All it does is set the started property to false.

And here's where the interesting behavior happens: when you try to drive the car, if it is started we get a "Zoom zoom!" and if not, we get a warning that we should start the car first.



Good catch. You're right; to start the car we could have replaced the code:

```
fiat.start();
```

with:

```
fiat.started = true;
```

That would have saved us from writing a method to start the car.

So why did we create and call the `start` method instead of just changing the `started` property directly? Using a method to change a property is another example of encapsulation whereby we can often improve the maintainability and extensibility of code by letting an object worry about how it gets things done. It's better to have a `start` method that knows how to start the car than for you to have to know "to start the car we need to take the `started` variable and set it to true."

Now you may still be saying "What's the big deal? Why not just set the property to true to start the car?!" Consider a more complex `start` method that checks the seatbelts, ensures there is enough fuel, checks the battery, checks the engine temperature and so on, all before setting `started` to true. You certainly don't want to think about all that every time you start the car. You just want a handy method to call that gets the job done. By putting all those details into a method, we've created a simple way for you to get an object to do some work while letting the object worry about how it gets that work done.

Take the fiat for a test drive

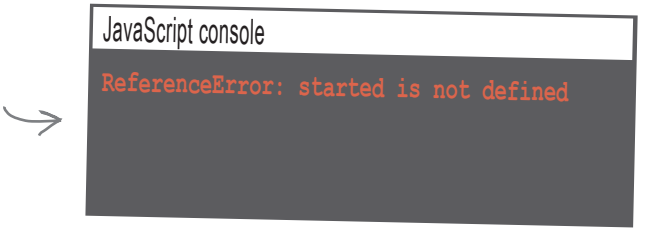
Let's take our new and improved `fiat` object for a test drive. Let's give it a good testing—we'll try to drive it before it's started, and then start, drive and stop it. To do that make sure you have the code for the `fiat` object typed into a simple HTML page ("carWithDrive.html"), including the new methods `start`, `stop` and `drive`, and then add this code below the object:

```
fiat.drive();   <- First, we'll try to drive the car, which should
fiat.start();   <- give us a message to start the car. Then we'll
fiat.drive();   <- start it for real, and we'll drive it. Finally,
fiat.stop();    <- when we're done we'll stop the car.
```

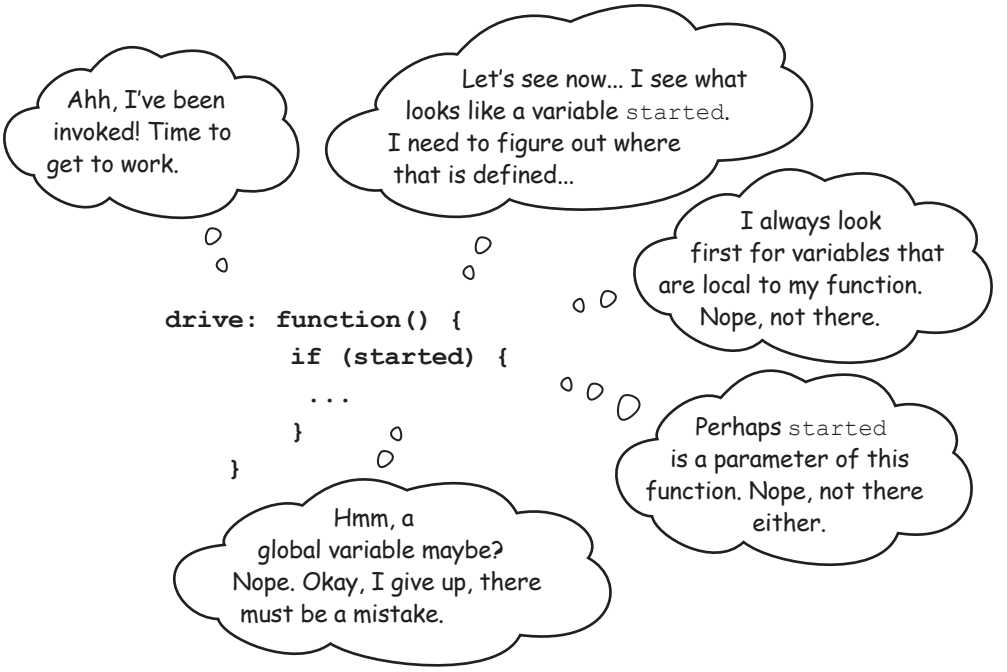
Go ahead and load the page in your browser and let the road trip begin!

Uh oh, not so fast...

If you can't drive your fiat, you're not alone. In fact, find your way to your JavaScript console and you're likely to see an error message similar to the one we got saying that `started` is not defined.



So, what's going on? Let's listen in on the `drive` method and see what's happening as we try to drive the car with `fiat.drive()`:



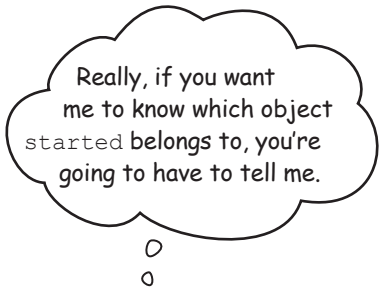
Why doesn't the drive method know about the started property?

Here's the conundrum: we've got references to the property `started` in the `fiat` object's methods, and normally when we're trying to resolve a variable in a function, that variable turns out to be a local variable, a parameter of the function or a global variable. But in the `drive` method, `started` is none of those things; instead, it's a *property* of the `fiat` object.

Shouldn't this code just work, though? In other words, we wrote `started` in the `fiat` object; shouldn't JavaScript be smart enough to figure out we mean the `started` property?

Nope. As you can see it isn't. How can that be?

Okay, here's the deal: what looks like a variable in the method is really a property of the object, but we aren't telling JavaScript which object. You might say to yourself, "Well, obviously we mean THIS object, this one right here! How could there be any confusion about that?" And, yes, we want the property of this very object. In fact, there's a keyword in JavaScript named `this`, and that is exactly how you tell JavaScript you mean *this object we're in*.




```
drive: function() {
         if (started) {
           ...
         }
     }
```

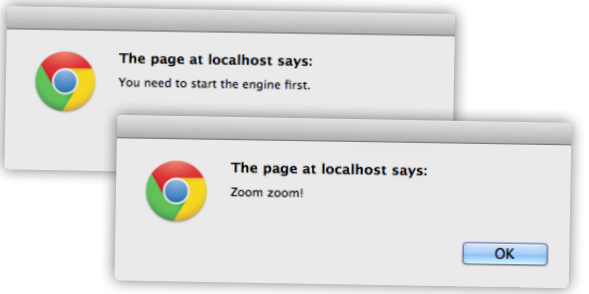
So, let's add the `this` keyword and get this code working:

```
var fiat = {
    make: "Fiat",
    // other properties are here, we're just saving space
    started: false,

    start: function() {
        this.started = true;
    },

    stop: function() {
        this.started = false;
    },

    drive: function() {
        if (this.started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Use this along with dot notation before each occurrence of the started property to tell the JavaScript interpreter you mean the property of THIS very object, rather than having JavaScript think you're referring to a variable.

A test drive with "this"

Go ahead and update your code, and take it for a spin! Here's what we got:



BE the Browser

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. After you've done the exercise look at the end of the chapter to see if you found them all.

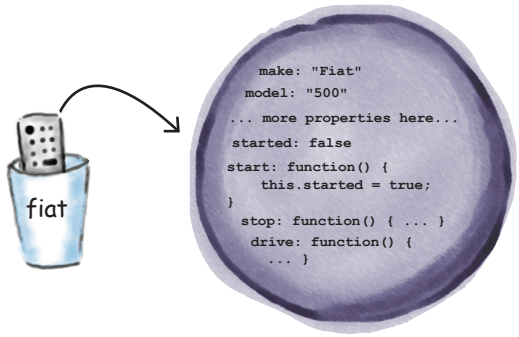
Go ahead and mark up the code right here...

```
var song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,

    play: function() {
        if (!playing) {
            this = true;
            console.log("Playing "
                + name + " by " + artist);
        }
    },

    pause: function() {
        if (playing) {
            this.playing = false;
        }
    }
};

this.play();
this.pause();
```

How this works

You can think of this like a variable that is assigned to the object whose method was just called. In other words, if you call the fiat object's start method, with fiat.start(), and use this in the body of the start method, then this will refer to the fiat object. Let's look more closely at what happens when we call the start method of the fiat object.

First, we have an object representing the Fiat car, which is assigned to the fiat variable:



Here's the fiat object with all its property names and values, including the start method.

Then, when we call the start method, JavaScript takes care of assigning this to the fiat object.

```
fiat.start();
```

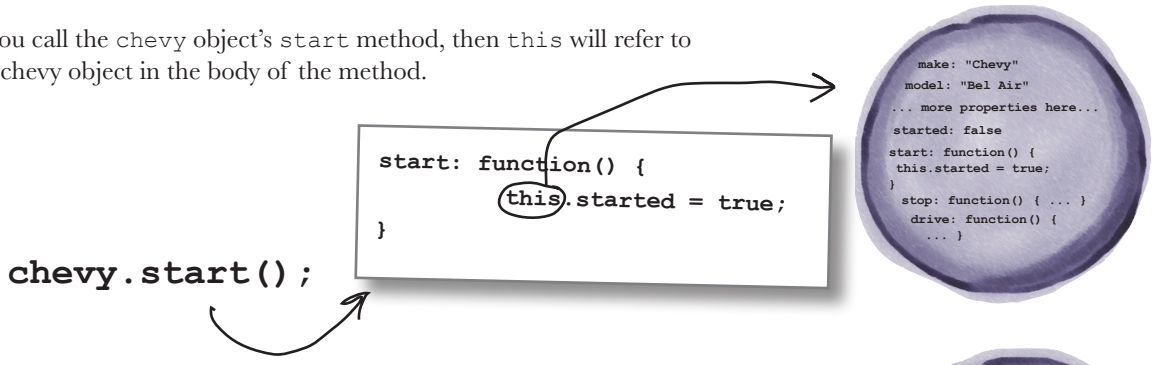
```
start: function() {
         this.started = true;

}
```

In this case, this refers to the fiat object, because we called the fiat object's start method.

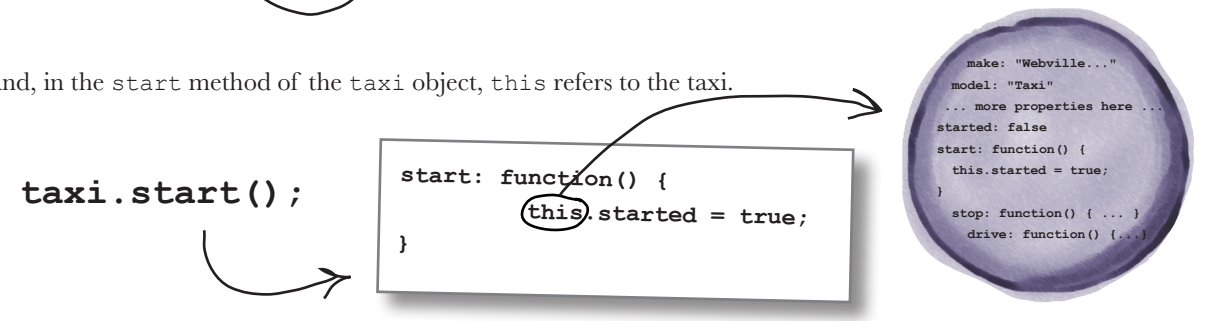
Whenever we call a method in an object, this will refer to that object. So here, this refers to the fiat object.

The real key to understanding this is that whenever a method is called, in the body of that method you can count on this to be assigned to the *object whose method was called*. Just to drive the point home, let's try it on a few other objects…

If you call the `chevy` object's `start` method, then `this` will refer to the chevy object in the body of the method.



And, in the `start` method of the `taxi` object, `this` refers to the taxi.



Sharpen your pencil

Use your new **this** skills to help us finish this code. Check your answer at the end of the chapter.

```
var eightBall = { index: 0,
                  advice: ["yes", "no", "maybe", "not a chance"],
                  shake: function() {
                             this.index = _________.index + 1;
                             if (_________.index  >= _________.advice.length) {
                                 _________.index = 0;
                             }
                  },
                  look: function() {
                             return _________.advice[_________.index];
                  }
};
eightBall.shake();
console.log(eightBall.look());
```

Repeat this sequence several times to test your code.

JavaScript console

```
no
maybe
not a chance
```

there are no Dumb Questions

Q: What's the difference between a method and a function?

A: A method is just a function that's been assigned to a property name in an object. You call functions using the function name, while you call methods using the object dot notation and the name of the property. You can also use the keyword this in a method to refer to the object whose method was called.

Q: I noticed that when using the function keyword within an object we don't give the function an explicit name. What happened to the function name?

A: Right. To call methods, we use the property name in the object rather than explicitly naming the function, and using that name. For now, just take this as the convention we use, but later in the book we'll dive into the topic of anonymous functions (which is what you call functions that don't explicitly have names).

Q: Can methods have local variables, like functions can?

A: Yes. A method is a function. We just call it a method because it lives inside an object. So, a method can do anything a function can do precisely because a method is a function.

Q: So, you can return values from methods too?

A: Yes. What we said in the last answer!

Q: What about passing arguments to methods? Can we do that too?

A: Err, maybe you didn't read the answer two questions back? Yes!

Q: Can I add a method to an object after it's created like I can with a property?

A: Yes. Think of a method as a function assigned to a property, so you can add a new one at any time:

```
// add a turbo method
car.engageTurbo =
        function() { ... };
```

Q: If I add a method like engageTurbo above, will the `this` keyword still work?

A: Yes. Remember `this` is assigned to the object whose method is called *at the time it is called*.

Q: When is the value of `this` set to the object? When we define the object, or when we call the method?

A: The value of `this` is set to the object when you call the method. So when you call fiat.start(), `this` is set to fiat, and when you call chevy.start(), `this` is set to chevy. It *looks* like `this` is set when you define the object, because in fiat.start, `this` is always set to fiat, and in chevy.start, `this` is always set to chevy. But as you'll see later, there is a good reason the value of `this` is set when you call the method and not when you define the object. This is an important point we'll be coming back to a few different times.

If you copy the start, stop, and drive methods into the chevy and cadi objects we created earlier, what do you have to change to make the methods work correctly?

Answer: Nothing! this refers to "this object," the one whose method we're calling.

It's time to get the whole fleet up and running. Add the drive method to each car object. When you've done that, add the code to start, drive and stop each of them. Check your answer at the end of the chapter.

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};
```

```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};
```

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

Don't forget to add a comma after mileage when you add the new properties!

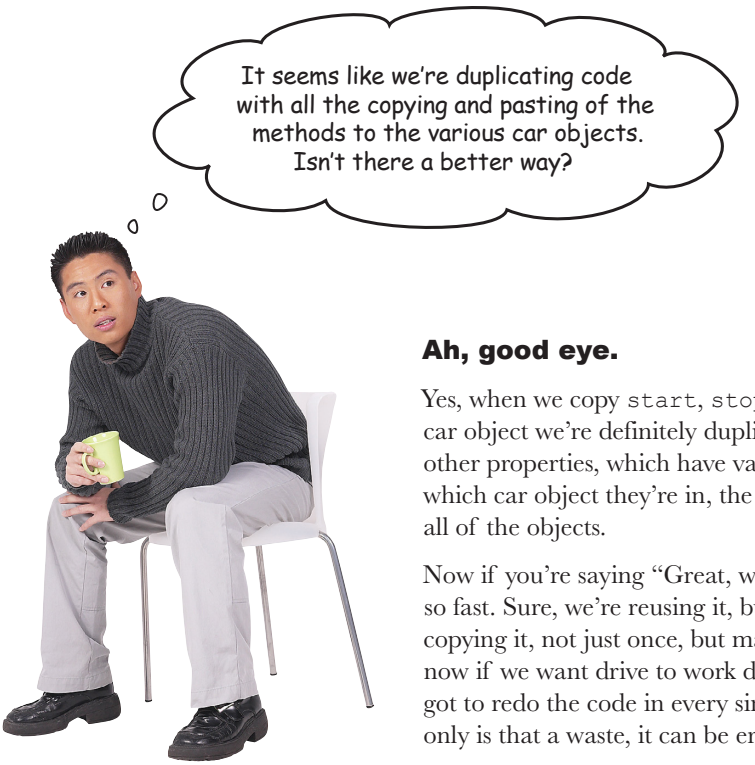
Add the started property and the methods to each car. Then use the code below to give them a test drive.

```
started: false,

start: function() {
   this.started = true;
},

stop: function() {
   this.started = false;
},

drive: function() {
   if (this.started) {
     alert(this.make + " " +
           this.model + " goes zoom zoom!");
   } else {
     alert("You need to start the engine first.");
   }
}
```

We improved the drive method just a bit so make sure you get this new code.

Throw this code after the car object definitions to give them all a test drive.

```
cadi.start();
cadi.drive();
cadi.stop();
chevy.start();
chevy.drive();
chevy.stop();
taxi.start();
taxi.drive();
taxi.stop();
```



Ah, good eye.

Yes, when we copy `start`, `stop` and `drive` into each car object we're definitely duplicating code. Unlike the other properties, which have values that depend on which car object they're in, the methods are the same for all of the objects.

Now if you're saying "Great, we're reusing code!"... not so fast. Sure, we're reusing it, but we're doing that by copying it, not just once, but many times! What happens now if we want drive to work differently? Then you've got to redo the code in every single car. Not good. Not only is that a waste, it can be error prone.

But you're identifying a problem even larger than simple copying and pasting; we're assuming that just because we put the same properties in all our objects, that makes them all car objects. What if you accidentally leave out the `mileage` property from one of the objects—is it still a car?

These are all real problems with our code so far, and we're going to tackle all these questions in an upcoming chapter on advanced objects where we'll talk about some techniques for properly reusing the code in your objects.

One thing you can do is iterate through an object's properties. To do that you can use a form of iteration we haven't seen yet called `for in`. The `for in` iterator steps through every property in an object in an arbitrary order. Here's how you could display all the properties of the `chevy` object:

`for in` steps through the object's properties one at time, assigning each one in turn to the variable prop.

```
for (var prop in chevy) {
    console.log(prop + ": " + chevy[prop]);
}
```

You can use prop as a way to access the property using bracket notation.

JavaScript console

```
make: Chevy
model: Bel Air
year: 1957
color: red
passengers: 2
convertible: false
mileage: 1021
```

This brings up another topic: there's another way to access properties. Did you catch the alternative syntax we just used to access the properties of the `chevy` object? As it turns out, you've got two options when accessing a property of an object. You already know dot notation:

```
chevy.color
```

We just use the object name followed by a dot and a property name.

But there's another way: bracket notation, which looks like this:

```
chevy["color"]
```

Here we use the object name followed by brackets that enclose a property name in quotes.

Looks a bit like how we access array items.

The thing to know about both of these forms, is they are equivalent and do the same thing. The only difference you need to know about is the bracket notation sometimes allows a little more flexibility because you can make the property name an expression like this:

```
chevy["co" + "lor"]
```

As long as the expression evaluates to a property name represented by a string, you can put any expression you want inside the backets.

How behavior affects state... Adding some Gas-o-line

Objects contain *state* and *behavior*. An object's properties allow us to keep state about the object—like its fuel level, its current temperature or, say, the current song that is playing on the radio. An object's methods allow us to have behavior—like starting a car, turning up the heat or fast-forwarding the playback of a song. Have you also noticed these two *interact*? Like, we can't start a car if it doesn't have fuel, and the amount of fuel should get reduced as we drive the car. Kinda like real life, right?

Let's play with this concept a little more by giving our car some fuel, and then we can start to add interesting behavior. To add fuel, we'll add a new property, `fuel`, and a new method, `addFuel`. The `addFuel` method will have a parameter, `amount`, which we'll use to increase the amount of fuel in the `fuel` property. So, add these properties to the `fiat` object:

```
var fiat = {
    make: "Fiat",
    model: "500",
    // other properties go here, we're saving some paper...
    started: false,
    fuel: 0,

    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
       if (this.started) {
         alert(this.make + " " + this.model + " goes zoom zoom!");
       } else {
         alert("You need to start the engine first.");
       }
    },
    addFuel: function(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

We've added a new property, fuel, to hold the amount of fuel in the car. The car will begin life on empty.

Let's also add a method, addFuel, to add fuel to the car. We can add as much fuel as we like by specifying the amount when we call the method.

Remember, fuel is an object property, so we need the this keyword...

But amount is a function parameter, so we don't need this to use it.

Now let's affect the behavior with the state

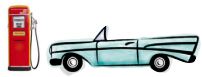
So now that we have fuel, we can start to implement some interesting behaviors. For instance, if there's no fuel, we shouldn't be able to drive the car! So, let's start by tweaking the `drive` method a bit to check the fuel level to make sure we've got some, and then we'll subtract one from `fuel` each time the car is driven. Here's the code to do that:

```
var fiat = {
    // other properties and methods here...
    drive: function() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
               this.fuel = this.fuel - 1;
            } else {
                alert("Uh oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel: function(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

Now we can check to make sure there's fuel before we drive the car. And, if we can drive the car, we should reduce the amount of fuel left each time we drive.

If there's no fuel left, we display a message and stop the engine. To drive the car again, you'll have to add fuel and restart the car.

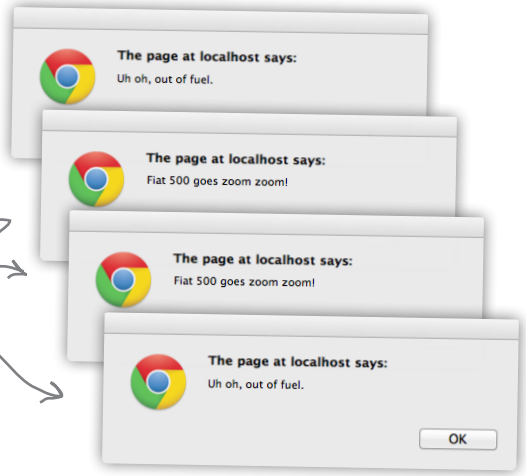
Gas up for a test drive



Go ahead and update your code, and take it for a spin! Here's what we got with the following test code:

```
fiat.start();
fiat.drive();
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.drive();
fiat.drive();
fiat.stop();
```

First, we tried to drive it with no fuel, so then we added some fuel and drove it until we ran out of fuel again! Try adding your own test code and make sure it works like you think it should.



We still have some more work to do to fully integrate the fuel property into the car. For instance, should you be able to start the car if there's no fuel? Check out the start method:

```
start: function() {
      this.started = true;
}
```

It certainly looks like we can.

Help us integrate the fuel property into this code by checking the fuel level before the car is started. If there's no fuel, and the start method is called, let the driver know with a handy alert, like `"The car is on empty, fill up before starting!"` Rewrite your start method below, and then add it to your code and test it. Check your answer at the end of the chapter before you go on.

Your code here.

Take a look at all the fiat car code. Are there other places you could use the fuel property to alter the car's behavior (or create behavior to modify the fuel property)? Jot down your ideas below.

Congrats on your first objects!

You've made it through the first objects chapter and you're ready to move forward. Remember how you began with JavaScript? You were thinking of the world in terms of low-level numbers and strings and statements and conditionals and for loops and so on. Look how far you've come. You're starting to think at a higher level, and in terms of objects and methods. Just look at this code:

```
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.stop();
```

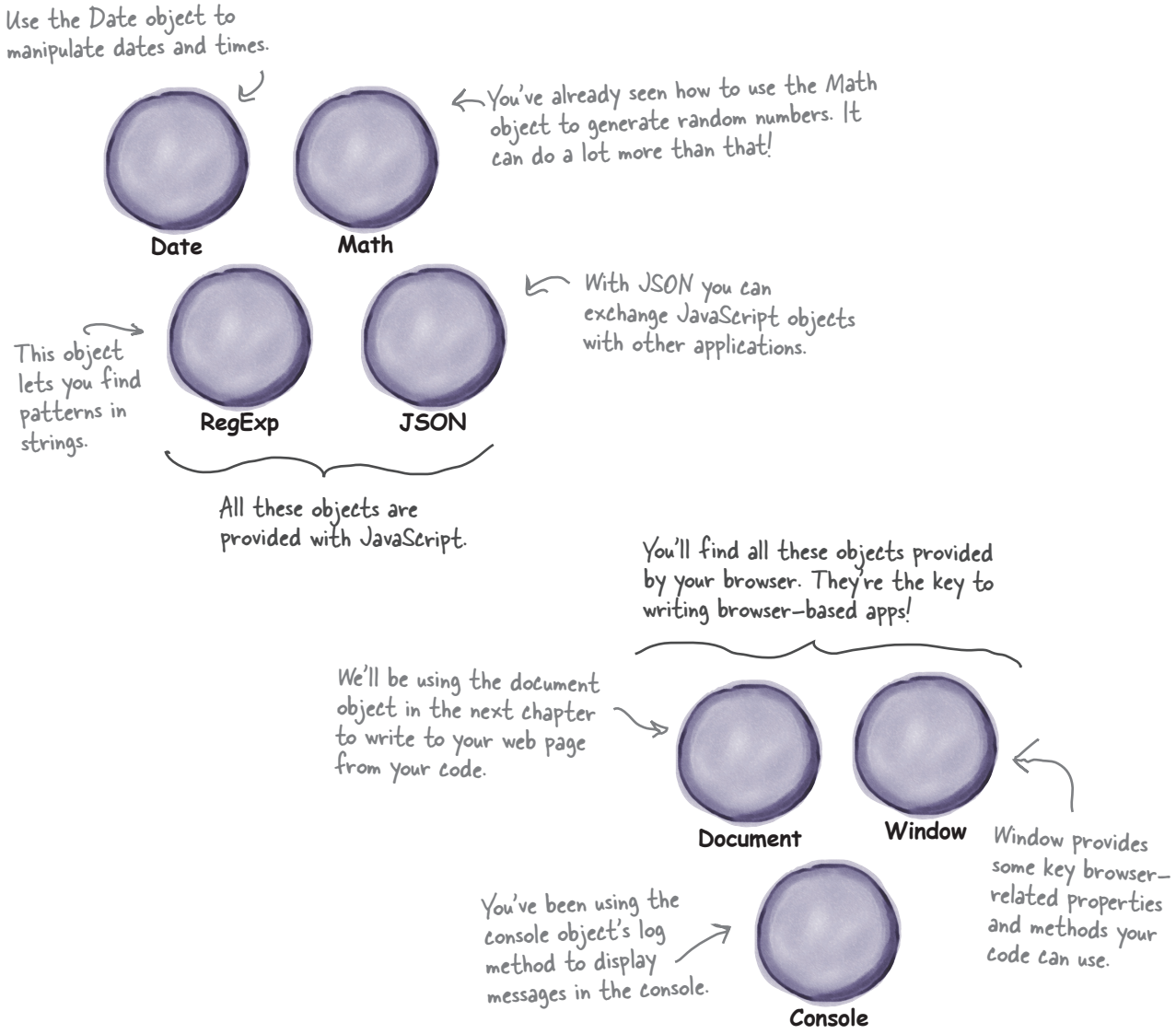
It's so much easier to understand what's going on in this code, because it describes the world as a set of objects with state and behavior.

And this is just the beginning. You can take it so much further, and we will. Now that you know about objects we're going to keep developing your skills to write truly object-oriented code using even more features of JavaScript and quite a few best practices (which become oh-so-important with objects).

There's one more thing you should know, before you leave this chapter…

Guess what? There are objects all around you! (and they'll make your life easier)

Now that you know a bit about objects, a whole new world is going to open up for you because JavaScript provides you with lots of objects (for doing math computations, manipulating strings and creating dates and times, to name a few) that you can use in your own code. JavaScript also provides some really key objects that you need to write code for the browser (and we're going to take a look at one of those objects in the next chapter). For now, take a second to get acquainted with a few more of these objects, and we'll touch on these throughout the rest of the book:



The Object Exposed

This week's interview: In Object's own words...

Head First: Welcome Object, it's been a fascinating chapter. It's a real head-spinner thinking about code as objects.

Object: Oh, well… we've only just begun.

Head First: How so?

Object: An object is a set of properties, right? Some of those properties are used to keep the state of the object, and some are actually functions—or rather, methods—that give an object behavior.

Head First: I'm with you so far. I hadn't actually thought about the methods being properties too, but I guess they are just another name and value, if you can call a function a value?

Object: Oh you can! Believe me, you can. In fact, that's a huge insight, whether you realize it or not. Hold on to that thought; I'm guessing there's a lot in store for you on that topic.

Head First: But you were saying…

Object: So, you've looked at these objects with their properties and you've created lots of them, like a bunch of different types of cars.

Head First: Right…

Object: But it was very *ad hoc*. The real power comes when you can create a template of sorts, something that can basically stamp out uniform objects for you.

Head First: Oh, you mean objects that all have the same type?

Object: Sort of… as you'll see the concept of type is an interesting one in JavaScript. But you're on the right track. You'll see that you have real power when you can start to write code that deals with objects of the same kind. Like you could write code that deals with vehicles and you wouldn't have to care if they are bicycles, cars or buses. That's power.

Head First: It certainly sounds interesting. What else do we need to know to do that?

Object: Well, you have to understand objects a little better, and you need a way to create objects of the same kind.

Head First: We just did that, didn't we? All those cars?

Object: They're sort of the same kind by convention, because you happened to write code that creates cars that look alike. In other words, they have the same properties and methods.

Head First: Right, and in fact we talked a little about how we are replicating code across all those objects, which is not necessarily a good thing in terms of maintaining that code.

Object: The next step is to learn how to create objects that really are all guaranteed to be the same, and that make use of the same code—code that's all in one place. That's getting into how to design object-oriented code. And you're pretty much ready for that now that you know the basics.

Head First: I'm sure our readers are happy to hear that!

Object: But there are a few more things about objects to be aware of.

Head First: Oh?

Object: There are many objects already out there in the wild that you can use in your code.

Head First: Oh? I hadn't noticed, where?

Object: How about `console.log`. What do you think `console` is?

Head First: Based on this discussion, I'm guessing it's an object?

Object: BINGO. And `log`?

Head First: A property… err, a method?

Object: BINGO again. And what about `alert`?

Head First: I haven't a clue.

Object: It has to do with an object, but we'll save that for a bit later.

Head First: Well, you've certainly given us a lot to think about Object, and I'm hoping you'll join us again.

Object: I'm sure we can make that work.

Head First: Great! Until next time then.

Crack the Code Challenge

In his quest for world domination, Dr. Evel has accidentally exposed an internal web page with the current passcode to his operation. With the passcode we can finally get the upper hand. Of course, as soon as Dr. Evel discovered the page was live on the Internet, he quickly took it down. Luckily, our agents made a record of the page. The only problem is, our agents don't know HTML or JavaScript. Can you help figure out the access code using the code below? Keep in mind, if you are wrong, it could be quite costly to Queen and Country.

TOP SECRET

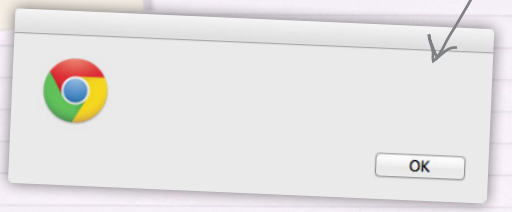
```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Here's the JavaScript.

Looks like this code is using a document object.

What pass code will you see in the alert?

Write your answer in the alert dialog box below.



Here's the HTML.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

The above JavaScript code is being included here.

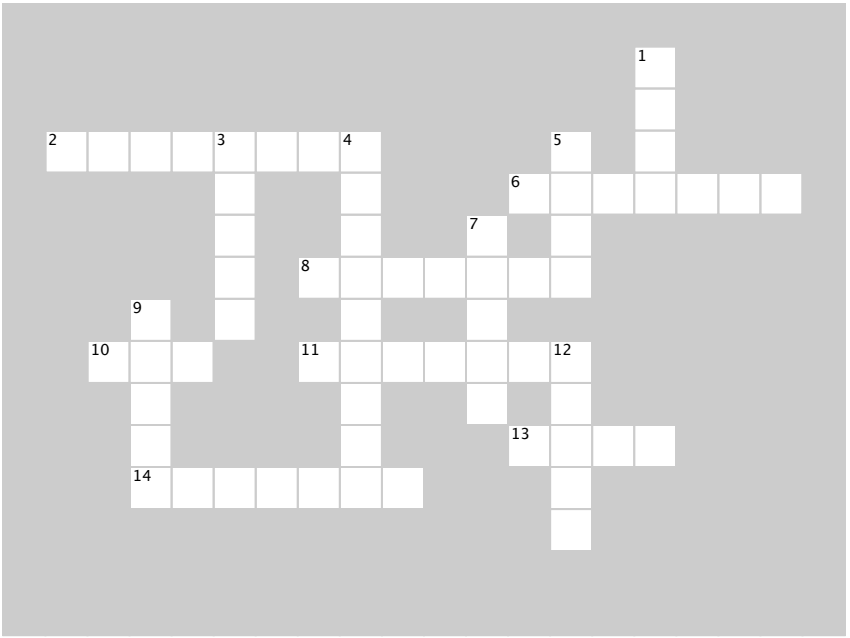
If you skipped the last page, go back and do the challenge. It is vitally important to Chapter Six!

BULLET POINTS

- An object is a **collection of properties**.
- To access a property, use **dot notation**: the name of the variable containing the object, then a period, then the name of the property.
- You can add new properties to an object at any time, by assigning a value to a new property name.
- You can also delete properties from objects, using the **delete** operator.
- Unlike variables that contain primitive values, like strings, numbers, and booleans, a variable can't actually contain an object. Instead, it contains a **reference** to an object. We say that objects are "reference variables".
- When you pass an object to a function, the function gets a copy of the reference to the object, not a copy of the object itself. So, if you change the value of one of the object's properties, it changes the value in the original object.
- Object properties can contain functions. When a function is in an object, we call it a method.
- You call a method by using the **dot notation**: the object name, a period, and the property name of the method, followed by parentheses.
- A method is just like a function except that it is in an object.
- You can pass arguments to methods, just like you can to regular functions.
- When you call an object's method, the keyword **this** refers to the object whose method you are calling.
- To access an object's properties in an object's method, you must use dot notation, with **this** in place of the object's name.
- In object-oriented programming, we think in terms of objects rather than procedures.
- An object has both **state** and **behavior**. State can affect behavior, and behavior can affect state.
- Objects **encapsulate**, or hide, the complexity of the state and behavior in that object.
- A well-designed object has methods that abstract the details of how to get work done with the object, so you don't have to worry about it.
- Along with the objects you create, JavaScript has many built-in objects that you can use. We'll be using many of these built-in objects throughout the rest of the book.

JavaScript cross

How about a crossword object? It's got lots of clue properties that will help objects stick in your brain.



ACROSS

2. An object gets _____ with its methods.

6. The method log is a property in the _____ object.

8. **this** is a _____, not a regular variable.

10. To access the property of an object we use ____ notation.

11. _____ can have local variables and parameters, just like regular functions can.

13. We used a _____ property to represent the make of a car object.

14. The _____ method affects the state of the car object, by adding to the amount of fuel in the car.

DOWN

1. The fiat wouldn't start because we weren't using _____ to access the started property.

3. Object references are passed by _____ to functions, just like primitive variables.

4. When you assign an object to a variable, the variable contains a _____ to the object.

5. We usually use one _____ for property names.

7. The name and value of a property in an object are separated by a _____.

9. Don't forget to use a _____ after each property value except the last one.

12. Car and dog objects can have both _____ and behavior.

Sharpen your pencil Solution

We've started making a table of property names and values for a car. Can you help complete it? Here's our solution:

Put your property names here.

And put the corresponding values over here.

```
{
  make         : "Chevy",
  model        : "Bel Air",
  year         : 1957,
  color        : "red",
  passengers   : 2,
  convertible  : false,
  mileage      : 1021,
  accessories  : "Fuzzy Dice",
  whitewalls   : true
};
```

We're using strings, booleans and numbers where appropriate.

Put your answers here. Feel free to expand the list to include your own properties.

Sharpen your pencil Solution

Use your new **this** skills to help us finish this code. Here's our solution.

```
var eightBall = { index: 0,
                  advice: ["yes", "no", "maybe", "not a chance"],
                  shake: function() {
                             this.index = this.index + 1;
                             if (this.index  >= this.advice.length) {
                                this.index = 0;
                             }
                  },
                  look: function() {
                             return this.advice[this.index];
                  }

};
eightBall.shake();
console.log(eightBall.look());
```

Repeat this sequence several times to test your code.

JavaScript console

```
no
maybe
not a chance
```

You don't have to be stuck with just one object. The real power of objects (as you'll see soon enough) is having lots of objects and writing code that can operate on whatever object you give it. Try your hand at creating another object from scratch… another car object. Go ahead and work out the code for your second object. Here's our solution.

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};
```

Here are the properties for the Cadillac.

We'll call this a tan color.

This is a 1955 GM Cadillac.

It's not a convertible, and it can hold five passengers (It's got a nice big bucket seat in the back).

Its mileage is 12,892.

Object Magnets Solution

Practice your object creating and dot notation skills by completing the code below with the fridge magnets. Be careful, some extra magnets got mixed in! Here's our solution.

```
 name: "Fido"
 weight: 20.2
age: 4
 breed: "mixed"
 activity: "fetch balls"
```

The dog object.

,
20.2
Fido
bark
age

Leftover magnets

Fido is hoping you get all his properties right.

```
var dog = {
    name: "Fido",
    weight: 20.2,
    age: 4,
    breed: "mixed",
    activity: "fetch balls"
};
var bark;
if (dog.weight > 20) {
   bark = "WOOF WOOF";
} else {
   bark = "woof woof";
}
var speak = dog.name + " says " + dog.bark + " when he wants to " + dog.activity;
console.log(speak);
```

Sharpen your pencil Solution

Your turn. Here are three more car objects; what is the result of passing each car to the `prequal` function? Work the answer by hand, and then write the code to check your answers. Here's our solution:

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};

prequal(cadi);
```

false

Write the value of prequal here.

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

false

```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};

prequal(chevy);
```

true

Sharpen your pencil Solution

You've been given a super secret file and two functions that allow access to get and set the contents of the file, but only if you have the right password. The first function, `getSecret`, returns the contents of the file if the password is correct, and logs each attempt to access the file. The second function, `setSecret`, updates the contents of the file, and resets the access tracking back to 0. It's your job to fill in the blanks below to complete the JavaScript, and test your functions. Here's our solution.

```
function getSecret(file, secretPassword) {
    file.opened = file.opened + 1;
    if (secretPassword == file.password) {
        return file.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == file.password) {
        file.opened = 0;
        file.contents = secret;
    }
}

var superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
var secret = getSecret(superSecretFile, 2);
console.log(secret);

setSecret(superSecretFile, 2, "Dr. Evel's next meeting is in Philadelphia.");
secret = getSecret(superSecretFile, 2);
console.log(secret);
```

The superSecretFile object is passed into the getSecret function, and gets the parameter name file. So we need to make sure we use the object name, file and dot notation to access the object's properties, like opened, and password.

Same here.

We can pass the supserSecretFile object to the getSecret and setSecret functions.

BE the Browser Solution

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. Here's our solution.

```
var song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,

    play: function() {
        if (!this.playing) {
            this.playing = true;
            console.log("Playing "
                + this.name + " by " + this.artist);
        }
    },

    pause: function() {
        if (this.playing) {
            this.playing = false;
        }
    }
};

this song.play();
this song.pause();
```

We were missing a this here.

And missing the playing property name here.

We need to use this to access both these properties, too.

Again here, we need this to access the playing property.

We don't use this outside of a method; we call an object using the object's variable name.

Exercise Solution

It's time to get the whole fleet up and running. Add the drive method to each car object. When you've done that, add the code to start, drive and stop each of them. Here's our solution.

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

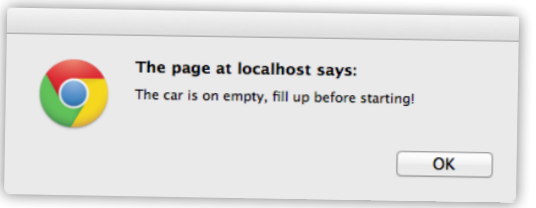
```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};

cadi.start();
cadi.drive();
cadi.stop();

chevy.start();
chevy.drive();
chevy.stop();

taxi.start();
taxi.drive();
taxi.stop();
```

Make sure you add a comma after any new properties you add.

We copied and pasted the code into each object, so every car has the same properties and methods.

Now we can start, drive and stop each of the cars, using the same method names.

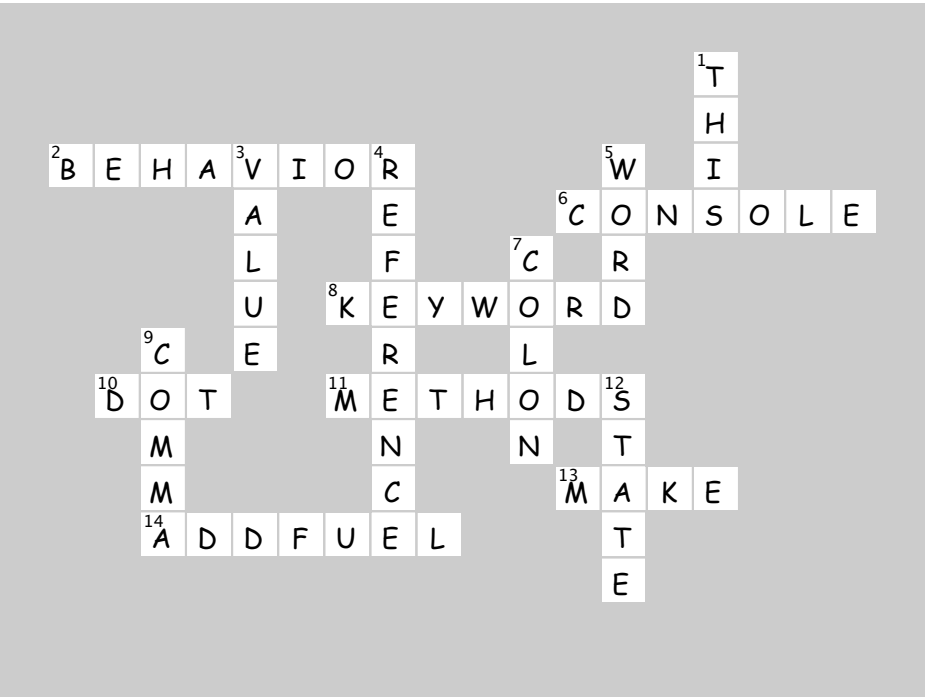
We still have some more work to do to fully integrate the fuel property into the car. For instance, should you really be able to start the car if there's no fuel? Help us integrate the fuel property into this code by checking the fuel level before the car is started. If there's no fuel, and the start method is called, let the driver know with a handy alert, like **`"The car is on empty, fill up before starting!"`** Rewrite the start method below, and then add it to your code and test it. Check your answer at the end of the chapter before you go on. Here's our solution.



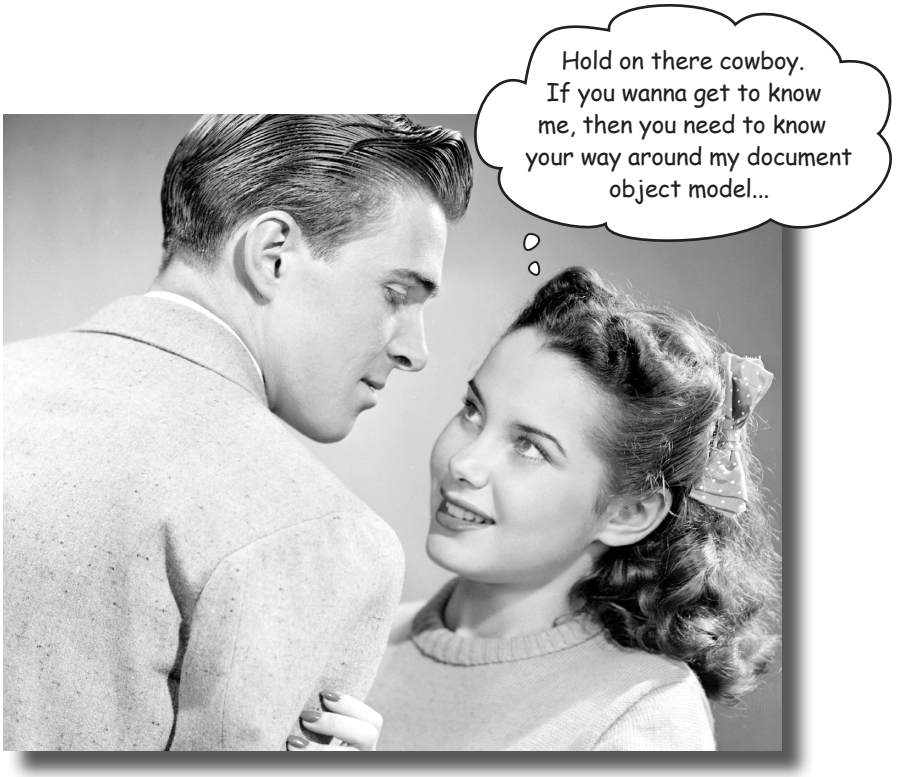

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    fuel: 0,
    started: false,

    start: function() {
        if (this.fuel == 0) {
            alert("The car is on empty, fill up before starting!");
        } else {
            this.started = true;
        }
    },

    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
                this.fuel = this.fuel - 1;
            } else {
                alert("Uh oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel: function(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

JavaScript cross Solution

How about a crossword object? It's got lots of clue properties that will help objects stick in your brain.



Getting to know the DOM



You've come a long way with JavaScript. In fact you've evolved from a newbie to a scripter to, well, a **programmer**. But, there's something missing. To really begin leveraging your JavaScript skills you need to know how to interact with the web page your code lives in. Only by doing that are you going to be able to write pages that are **dynamic**, pages that react, that respond, that update themselves after they've been loaded. So how do you interact with the page? By using the **DOM**, otherwise known as the **document object model**. In this chapter we're going to break down the DOM and see just how we can use it, along with JavaScript, to teach your page a few new tricks.

In our last chapter, we left you with a little challenge. The "crack the code challenge."

You were given some HTML with code in an external file, captured from Dr. Evel's web site, that looked like this:

Here's the HTML.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

Notice that each paragraph is identified by an id.

Here's the JavaScript....

```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

document is a global object.

And getElementById is a method.

Make sure you get the case right on the letters in the method name getElementById, otherwise it won't work!

And look, we have dot notation, this looks like an object with an innerHTML property.

And you needed to figure out Dr. Evel's passcode using your deductive powers on this code.

So what does the code do?

We'll learn all about document and element objects in this chapter.

Let's walk through this code to see how Dr Evel is generating his passcodes. After we break down each step you'll start to see how this all works:

1. First, the code sets the variable `access` to the result of calling the `document` object's `getElementById` method and passing it "code9". What gets returned is an element object.

```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Get the element that has an id of "code9". That would be this element...

```
<p id="code9">My watch stopped at</p>
```

---

2. Next we take that element (that is, the element with the id "code9") and we use its `innerHTML` property to get its content, which we assign to the variable `code`.

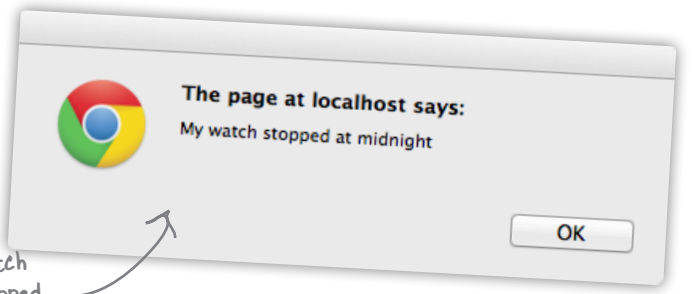
```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

The element with id "code9" is a paragraph element and that element's content (or rather its "innerHTML") is the text "My watch stopped at".

---

3. Dr. Evel's code adds the string " midnight" to the end of string contained in `code`, which is "My watch stopped at". Then, the page creates an alert with the passcode contained in the variable `code`.

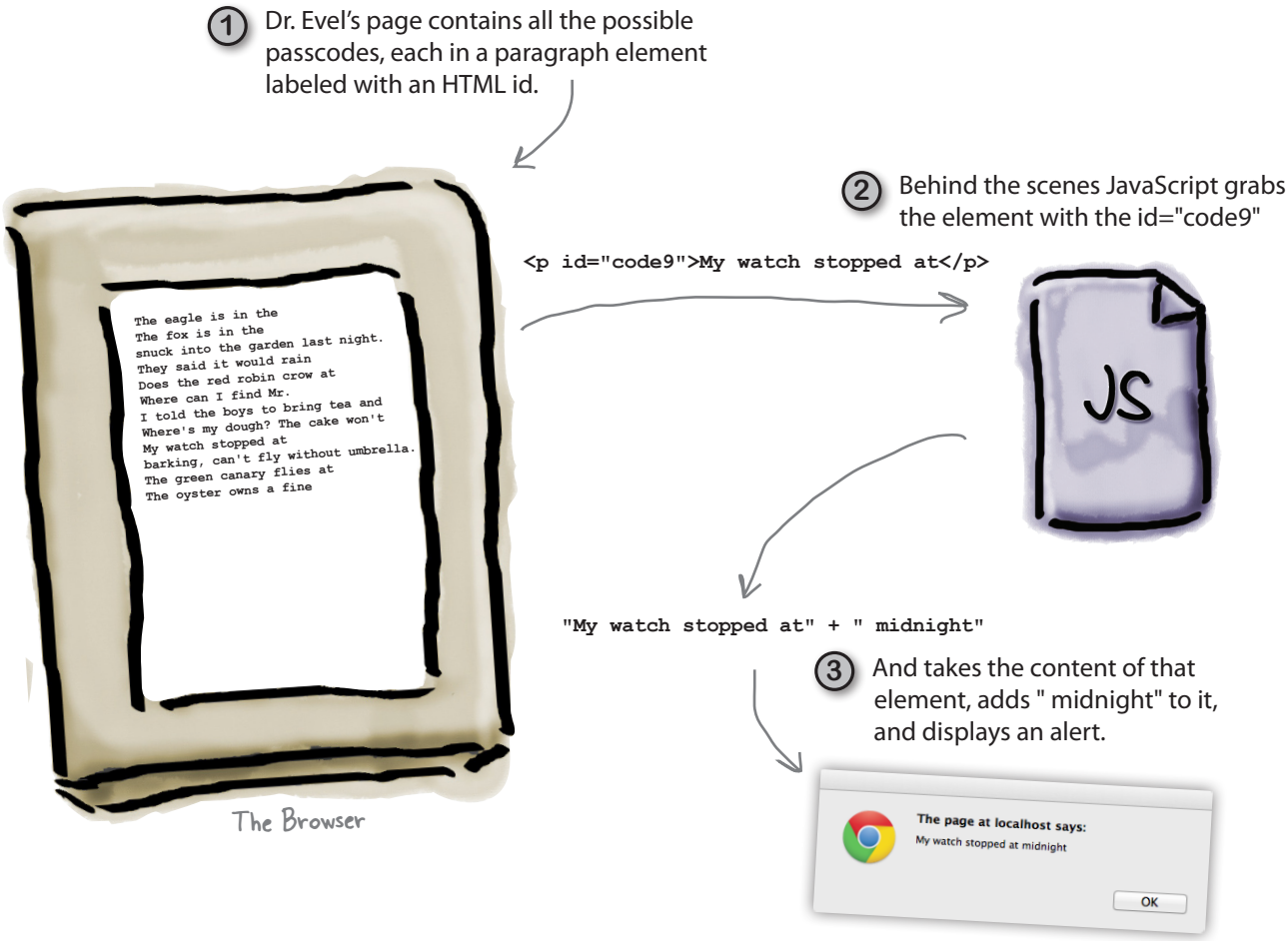
```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```



So we add " midnight" to "My watch stopped at" to get "My watch stopped at midnight" and then put up an alert to display this code.

A quick recap

So, what did we just do? Well, we had some JavaScript that reached into the page (otherwise known as the *document*), grabbed an element (the one with the id equal to `"code9"`), took that element's content (which is `"My watch stopped at"`), slapped a `" midnight"` on the end, and then displayed the result as a passcode.



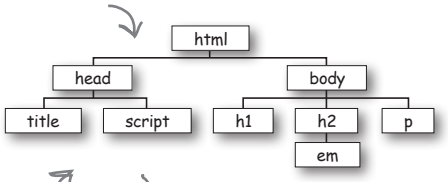
Now, more power to Dr. Evel and his JavaScript skills, and we wish him the best in his security schemes, but what is important here is to notice that the web page is a living, breathing *data structure* that your JavaScript can interact with—you can access and read the content of the elements in your page. You can also go the other way, and use JavaScript to change the content or structure of your page. To do all that, let's step back for a moment and understand better how JavaScript and HTML work together.

How JavaScript really interacts with your page

JavaScript and HTML are *two different things*. HTML is markup and JavaScript is code. So how do they interact? It all happens through a representation of your page, called the *document object model*, or the DOM for short. Where does the DOM come from? It's created when the browser loads your page. Here's how:

1 When you load a page into the browser, not only does the browser parse the HTML and then render it to the display, it also creates a set of objects that represent your markup. These objects are stored in the DOM.

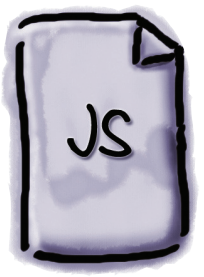
We call this the Document Object Model...



...or DOM, for short.

Your browser

2 Your JavaScript can interact with the DOM to get access to the elements and the content in them. JavaScript can also use the DOM to create or remove elements.



3 When JavaScript modifies the DOM, the browser updates the page dynamically, so you see new content on your page.

How to bake your very own DOM

Let's take some markup and create a DOM for it. Here's a simple recipe for doing that:

Ingredients

One well-formed HTML5 page

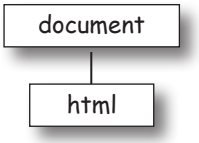
One modern web browser, pre-heated and ready to go

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My blog</title>
  <script src="blog.js"></script>
</head>
<body>
  <h1>My blog</h1>
  <div id="entry1">
    <h2>Great day bird watching</h2>
    <p>
      Today I saw three ducks!
      I named them
      Huey, Louie, and Dewey.
    </p>
    <p>
      I took a couple of photos...
    </p>
  </div>
</body>
</html>
```

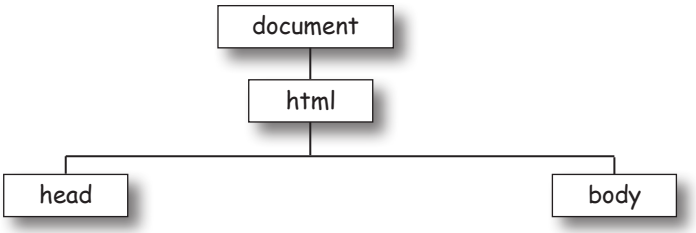
Instructions

1. Start by creating a document node at the top.

2. Next, take the top level element of your HTML page, in our case the <html> element, call it the current element and add it as a child of the document.



3. For each element nested in the current element, add that element as a child of the current element in the DOM.



4. Return to (3) for each element you just added, and repeat until you are out of elements.

A first taste of the DOM

If you follow the recipe for creating a DOM you'll end up with a structure like the one below. Every DOM has a document object at the top and then a tree complete with branches and leaf nodes for each element in the HTML markup. Let's take a closer look.

We compare this structure to a tree because a "tree" is a data structure that comes from computer science, and because it looks like an upside down tree, with the root at the top and the leaves at the bottom.

document is always at the top. document is a special part of the tree that you can use in JavaScript to get access to the entire DOM.

document is also like the root of an upside down tree.

These are like the branches of the tree.

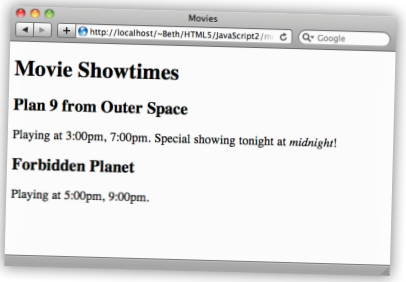
These are like the leaves of the tree (because there are no elements inside them, just text).

The DOM includes the content of the page as well as the elements. (We don't always show all the text content when we draw the DOM, but it's there).

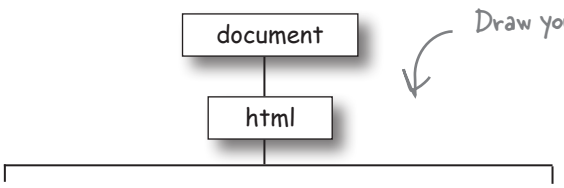
BE the Browser

Your job is to act like you're the browser. You need to parse the HTML and build your very own DOM from it. Go ahead and parse the HTML to the right, and draw your DOM below. We've already started it for you.

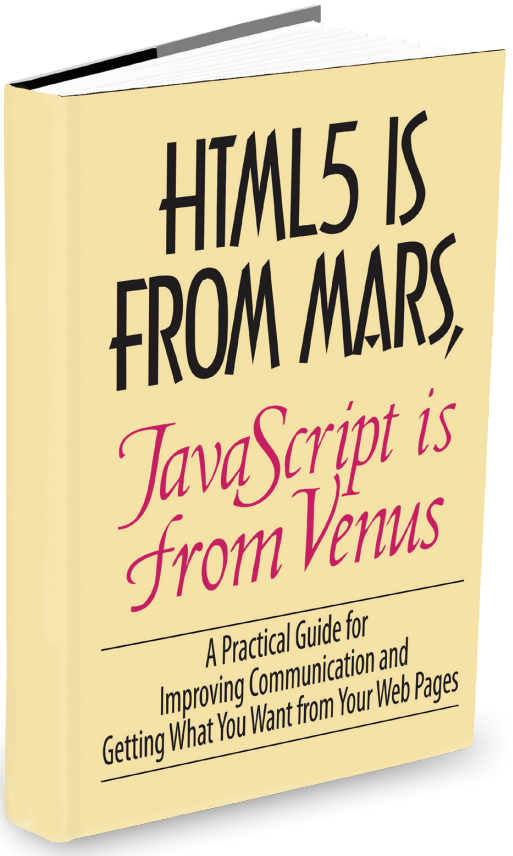
Check your answer with our solution at the end of the chapter before you go on.




```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Movies</title>
  </head>
  <body>
    <h1>Movie Showtimes</h1>
    <h2 id="movie1">Plan 9 from Outer Space</h2>
    <p>Playing at 3:00pm, 7:00pm.
      <span>
        Special showing tonight at <em>midnight</em>!
      </span>
    </p>
    <h2 id="movie2">Forbidden Planet</h2>
    <p>Playing at 5:00pm, 9:00pm.</p>
  </body>
</html>
```



Draw your DOM here.

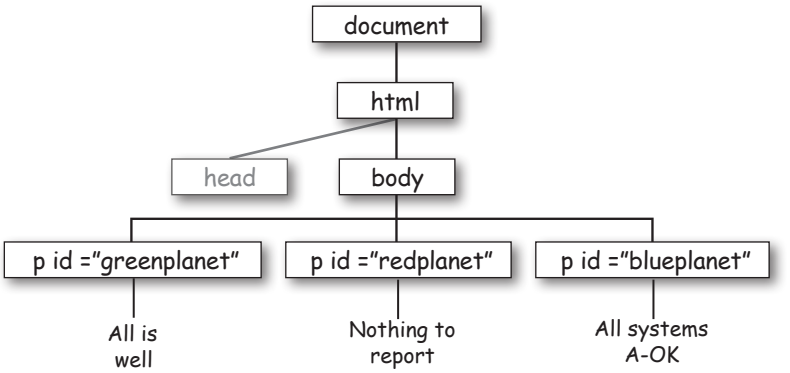


Or, how two totally different technologies hooked up.

HTML and JavaScript are from different planets for sure. The proof? HTML's DNA is made of declarative markup that allows you to describe a set of nested elements that make up your pages. JavaScript, on the other hand, is made of pure algorithmic genetic material, meant for describing computations.

Are they so far apart they can't even communicate? Of course not, because they have something in common: the DOM. Through the DOM, JavaScript can communicate with your page, and vice versa. There are a few ways to make this happen, but for now let's concentrate on one—it's a little wormhole of sorts that allows JavaScript to get access to any element in your page. That wormhole is *getElementById*.

Let's start with a DOM. Here's a simple DOM; it's got a few HTML paragraphs, each with an id identifying it as the green, red or blue planet. Each paragraph has some text as well. Of course there's a <head> element too, but we've left the details out to keep things simpler.

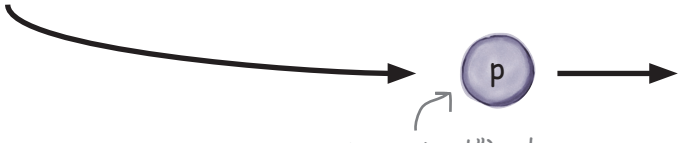


Now let's use JavaScript to make things more interesting. Let's say we want to change the greenplanet's text from "All is well" to "Red Alert: hit by phaser fire!" Down the road you might want to do something like this based on a user's actions, or even based on data from a web service. We'll get to all that; for now let's just get the greenplanet's text updated. To do that we need the element with the id "greenplanet". Here's some code that does that:

The document represents the entire page in your browser and contains the complete DOM, so we can ask it to do things like find an element with a specific id.

Here we're asking the document to get us an element by finding the element that matches the given id.

```
document.getElementById("greenplanet");
```



getElementById("greenplanet") returns the paragraph element corresponding to "greenplanet"...

...and then the JavaScript code can do all sorts of interesting things with it.

Once getElementById gives you an element, you're ready do something with it (like change its text to "Red Alert: hit by phaser fire!"). To do that, we typically assign the element to a variable so we can refer to the element thoughout our code. Let's do that and then change the text:

We're assigning the element to a variable named planet.

Here's our call to getElementById, which seeks out the "greenplanet" element and returns it.

```
var planet = document.getElementById("greenplanet");
```

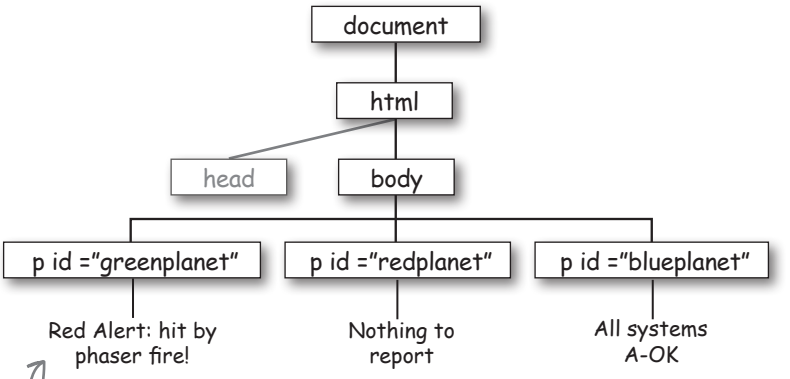
And in our code we can now just use the variable planet to refer to our element.

```
planet.innerHTML = "Red Alert: hit by phaser fire!";
```

We can use the innerHTML property of our planet element to change the content of the element.

We'll talk more about properties of elements shortly...

We change the content of the greenplanet element to our new text... which results in the DOM (and your page) being updated with the new text.

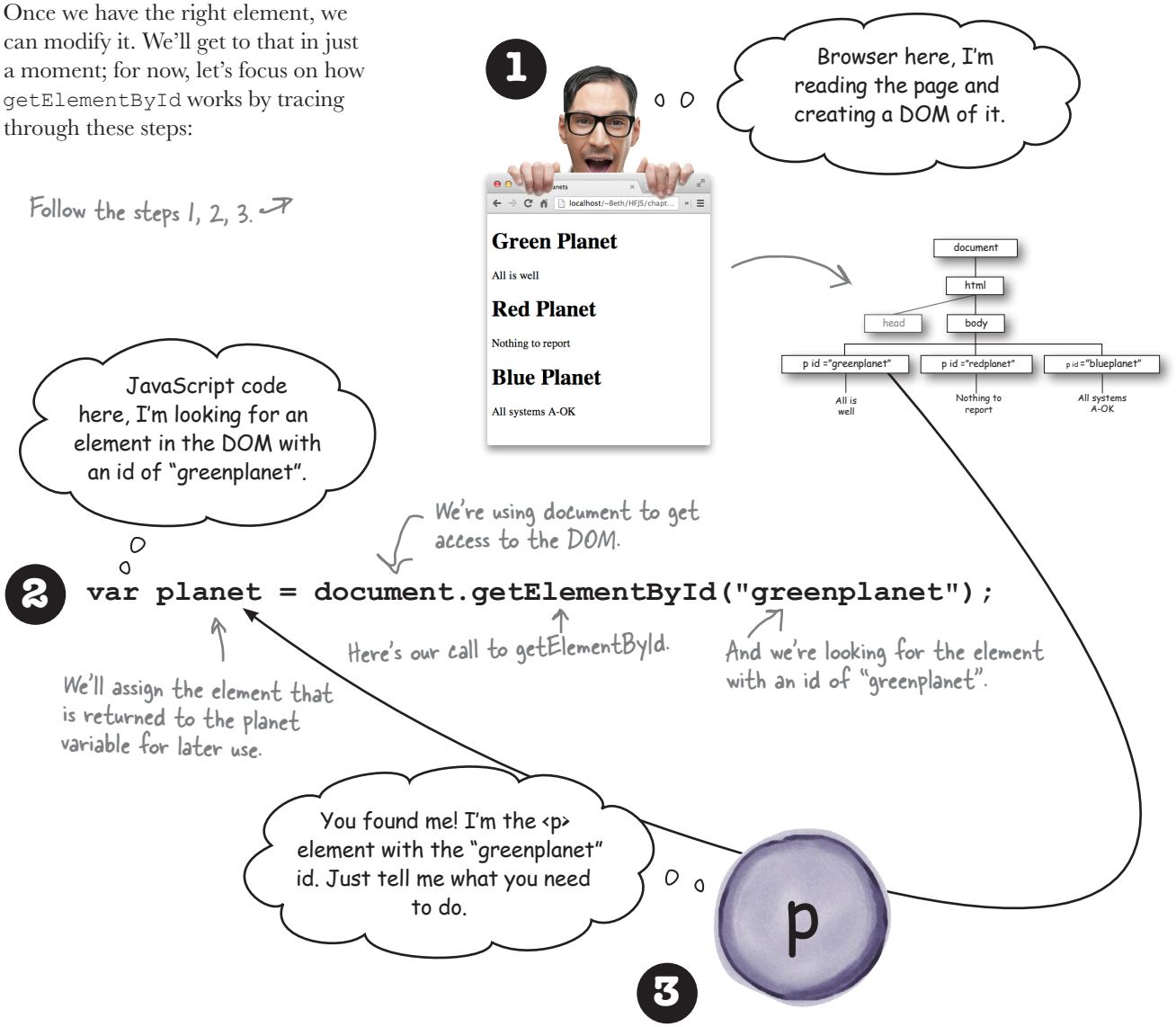


Any changes to the DOM are reflected in the browser's rendering of the page, so you'll see the paragraph change to contain the new content!

Getting an element with getElementById

So, what did we just do? Let's step through it in a little more detail. We're using the `document` object to get access to the DOM from our code. The `document` object is a built-in object that comes with a bunch of properties and methods, including `getElementById`, which we can use to grab an element from the DOM. The `getElementById` method takes an id and returns the element that has that id. Now in the past you've probably used ids to select and style elements with CSS. But here, what we're doing is using an id to grab an element—the <p> element with the id "greenplanet"—from the DOM.

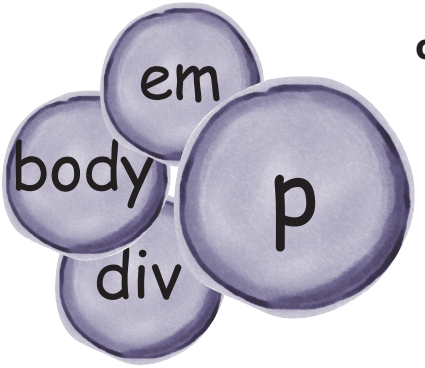
Once we have the right element, we can modify it. We'll get to that in just a moment; for now, let's focus on how `getElementById` works by tracing through these steps:



What, exactly, am I getting from the DOM?

When you grab an element from the DOM using `getElementById`, what you get is an *element object*, which you can use to read, change or replace the element's content and attributes. And here's the magic: when you change an element, *you change what is displayed in your page as well*.

But, first things first. Let's take another look at the element object we just grabbed from the DOM. We know that this element object represents the <p> element in our page that has the id "greenplanet" and that the text content in the element is "All is well". Just like other kinds of JavaScript objects, an element object has properties and methods. In the case of an element object, we can use these properties and methods to read and change the element. Here are a few things you can do with element objects:



Get the content (text or HTML).

Change the content.

Read an attribute.

Add an attribute.

Change an attribute.

Remove an attribute.

Things you can do with an element object.

What we want to do with our <p> element—which, remember, is the <p> element with the id "greenplanet"—is change the content "All is well" to "Red Alert: hit by phaser fire!". We've got the element object stashed in the `planet` variable in our code; let's use that to modify one of its properties, `innerHTML`:

The planet variable contains an element object–the element object that is the "greenplanet" <p> element.

```
var planet = document.getElementById("greenplanet");

planet.innerHTML = "Red Alert: hit by phaser fire!";
```

We can use the innerHTML property of the element object to change the content of the element!

Finding your inner HTML

The `innerHTML` property is an important property that we can use to read or replace the content of an element. If you look at the value of `innerHTML` then you'll see the content contained *within* the element, not including the HTML element tags. The "withIN" is why it's called "inner" HTML. Let's try a little experiment. We'll try displaying the content of the `planet` element object in the console by logging the `innerHTML` property. Here's what we get:

```
var planet = document.getElementById("greenplanet");
console.log(planet.innerHTML);
```

We're just passing the planet.innerHTML property to console.log to log to the console.

JavaScript console

```
All is well
```

The content of the innerHTML property is just a string, so it displays just like any other string in the console.

Now let's try changing the value of the `innerHTML` property. When we do this, we're changing the content of the "greenplanet" `<p>` element in the page, so you'll see your page change too!

```
var planet = document.getElementById("greenplanet");
planet.innerHTML = "Red Alert: hit by phaser fire!";
console.log(planet.innerHTML);
```

Now we're changing the content of the element by setting its innerHTML property to the string "Red Alert: hit by phaser fire!"

JavaScript console

```
Red Alert: hit by
phaser fire!
```

So when we log the value of the innerHTML property to the console we see the new value.

And the web page changes too!

A Quick Refresher

Hey, sit down; take a quick break. You might be saying to yourself, "Wait, I remember something about ids and classes but I don't remember the specifics, and don't they have something to do with CSS anyway?" No, problem, let's just have a quick refresher, get some context, and we'll have you back on your way in no time...

With HTML, ids give us a way to uniquely identify an element, and, once an element is unique, we can use that to select it with CSS for styling. And, as you've seen, we can get an element by its id in JavaScript as well.

Let's look at an example:

```
<div id="menu">
  ...
</div>
```

We're giving this a unique id of "menu". It should be the only element in your page with the id "menu".

And once we have that, we can select it with CSS to style it. Like this:

div#menu is an id selector.

```
div#menu {
    background-color: #aaa;
}
```

div#menu selects the with the id menu, so we can apply style to that element, and only that element.

And we can access this element through its id in JavaScript too:

```
var myMenu = document.getElementById("menu");
```

Don't forget, there's another way to label your elements: with classes. Classes give us a way to label a set of elements, like this:

```
<h3 class="drink">Strawberry Blast</h3>
<h3 class="drink">Lemon Ice</h3>
```

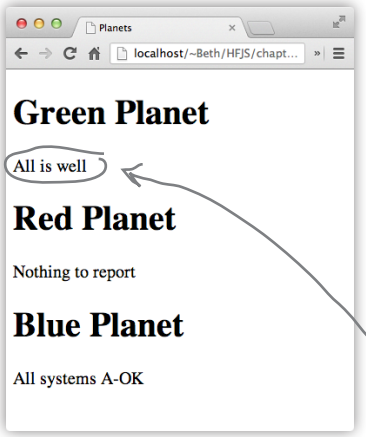
Both <h3> elements are in the class "drink". A class is like a group; you can have multiple elements in the same group.

And we can select elements by classes too, both in CSS and JavaScript. We'll see how to make use of classes with JavaScript in a bit. And, by the way, if this reminder isn't quite enough, check out Chapter 7 of *Head First HTML and CSS*, or your favorite HTML & CSS reference guide.

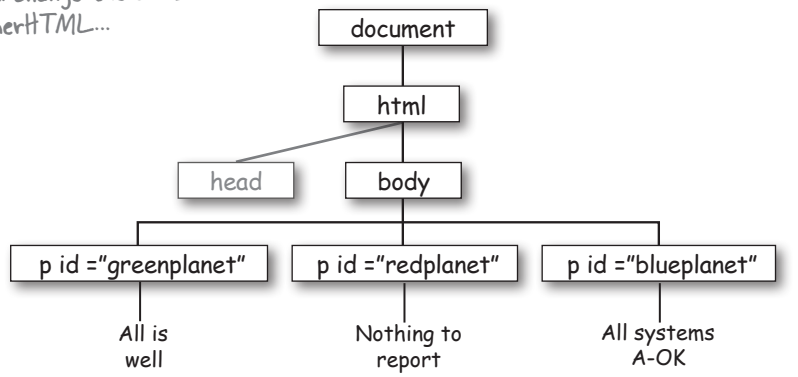
What happens when you change the DOM

So, what exactly happens when you change the content of an element using `innerHTML`? What you're doing is changing actual content of your web page, on the fly. And when you change the content in the DOM, you'll see that change immediately in your web page, too.

Before...



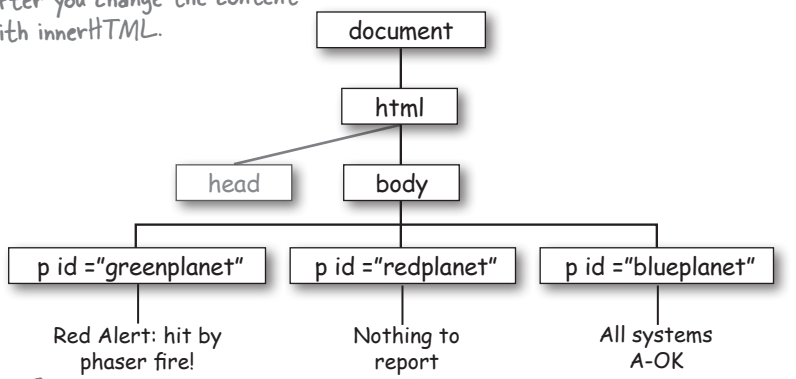
The web page you see and the DOM behind the scenes before you change the content with innerHTML...



This is the element whose content we're going to change...

... and after.

... and the web page you see and the DOM behind the scenes after you change the content with innerHTML.



Any changes to the DOM are reflected in the browser's rendering of the page, so you'll see the paragraph change to contain the new content!

there are no Dumb Questions

Q: What happens if I use document.getElementById and pass in an id that doesn't exist?

A: If you try to get an element from the DOM by id, and that id doesn't exist in an element, then the call to getElementById returns a null value. Testing for null is a good idea when you use getElementById to ensure that the element is there before you try to access its properties. We'll talk more about null in the next chapter.

Q: Can we use document.getElementById to get elements by class as well—for instance, say I have a bunch of elements in the class "planets"?

A: No, but you're thinking along the right lines. You can only use getElementById with an id. But there is another DOM method named getElementsByClassName that you can use to get elements by class name. With this method, what you get back is a collection of elements that belong to the class (because multiple elements can be in the same class). Another method that returns a collection of elements is getElementsByTagName, which returns all elements that match the tag name you specify. We'll see getElementsByTagName a little later in the book and see how to handle the collection of elements it returns.

Q: What exactly is an element object anyway?

A: Great question. An element object is the browser's internal representation of what you type into your HTML file, like some text . When the browser loads and parses your HTML file, it creates an element object for every element in your page, and adds all those element objects to the DOM. So the DOM is really just a big tree of element objects. And, keep in mind that, just like other objects, element objects can have properties, like innerHTML, and methods, too. We'll explore a few more of the properties and methods of element objects later in the book.

Q: I would have expected a property named "content" or maybe "html" in the element object. Why is it called innerHTML instead?

A: We agree, it's kind of a weird name. The innerHTML property represents all the content contained in your element, including other nested elements (like a paragraph might include and <img> elements in addition to the text in the paragraph). In other words, it's the HTML that's "INside" your element. Is there an outerHTML property? Yes! And that property gets you all the HTML inside the element, as well as the element itself. In practice you won't see outerHTML used very often, but you will see innerHTML used frequently to update the content of elements.

Q: So by assigning something to innerHTML I can replace the content of any element with something else. What if I used innerHTML to change, say, the <body> element's content?

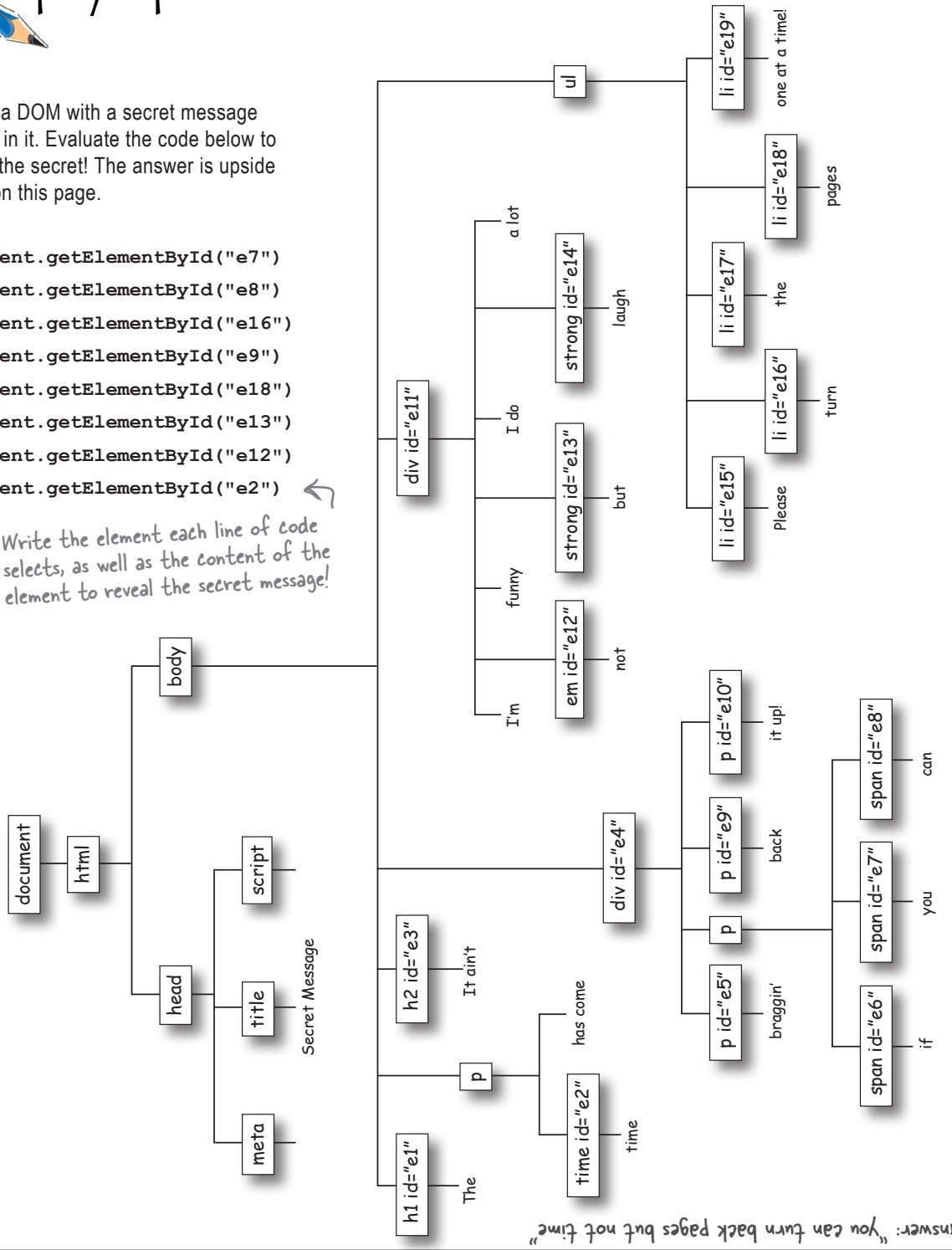
A: Right, innerHTML gives you a convenient way to replace the content of an element. And, yes, you could use it to replace the content of the <body> element, which would result in your entire page being replaced with something new.

Sharpen your pencil

Here's a DOM with a secret message hidden in it. Evaluate the code below to reveal the secret! The answer is upside down on this page.

```
document.getElementById("e7")
document.getElementById("e8")
document.getElementById("e16")
document.getElementById("e9")
document.getElementById("e18")
document.getElementById("e13")
document.getElementById("e12")
document.getElementById("e2")
```

Write the element each line of code selects, as well as the content of the element to reveal the secret message!



Answer: "You can turn back pages but not time."

A test drive around the planets

Okay, you know how to use `document.getElementById` to get access to an element, and how to use `innerHTML` to change the content of that element. Let's do it for real, now.

Here's the HTML for the planets. We've got a `<script>` element in the `<head>` where we'll put the code, and three paragraphs for the green, red, and blue planets. If you haven't already, go ahead and type in the HTML and the JavaScript to update the DOM:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planets</title>
  <script>
    var planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
  </script>
</head>
<body>
  <h1>Green Planet</h1>
  <p id="greenplanet">All is well</p>
  <h1>Red Planet</h1>
  <p id="redplanet">Nothing to report</p>
  <h1>Blue Planet</h1>
  <p id="blueplanet">All systems A-OK</p>
</body>
</html>
```

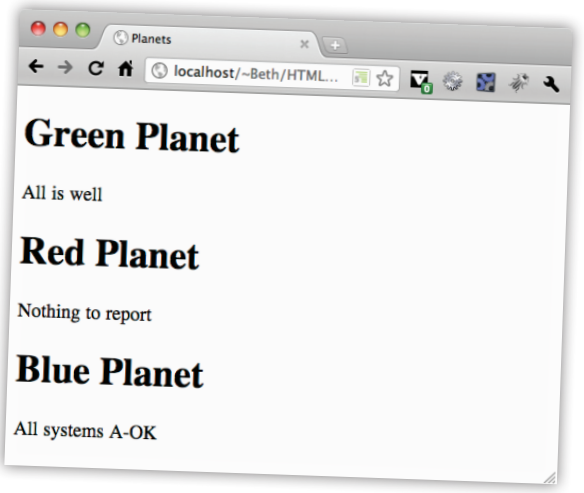
Here's our script element with the code.

Just like you saw before, we're getting the <p> element with the id "greenplanet" and changing its content.

Here's the <p> element you're going to change with JavaScript.

After you've got it typed in, go ahead and load the page into your browser and see the DOM magic happen on the green planet.

UH OH! Houston, we've got a problem, the green planet still shows "All is well". What's wrong?



I've triple-checked my markup and code, and this just isn't working for me either. I'm not seeing any changes to my page.

Oh yeah, we forgot to mention one thing.
When you're dealing with the DOM it's important to execute your code only *after* the page is *fully loaded*. If you don't, there's a good chance the DOM won't be created by the time your code executes.

Let's think about what just happened: we put code in the `<head>` of the page, so it begins executing before the browser even sees the rest of the page. That's a big problem because that paragraph element with an id of "greenplanet" doesn't exist, yet.

So what happens exactly? The call to `getElementById` returns null instead of the element we want, causing an error, and the browser, being the good sport that it is, just keeps moving and renders the page anyway, but without the change to the green planet's content.

Check out your console when this page loads, you'll see the error in most browsers. The console tool is good for debugging.

How do we fix this? Well, we could move the code to the bottom of the `<body>`, but there's actually a more foolproof way to make sure this code runs at the right time; a way to tell the browser "run my code after you've fully loaded in the page and created the DOM." Let's see how to do that next.

JavaScript console

```
Uncaught TypeError:
Cannot set property
'innerHTML' of null
```

Don't even think about running my code until the page is fully loaded!

Ah, but how? Besides moving the code to the bottom of the body, there's another—and, one might argue—cleaner way to do it: *with code*.

Here's how it works: first create a function that has the code you'd like executed *once the page is fully loaded*. After you've done that, you take the window object, and assign the function to its `onload` property.

The window object is built-in to JavaScript. It represents the browser window.

What does that do? The window object will call any function you've assigned to its `onload` property, but only *after* the page is fully loaded. So, thank the designers of the window object for giving you a way to supply the code that gets called after the page has loaded. Check this out:

You can call this function anything you want, but it's often called init by convention.

First, create a function named init and put your existing code in the function.

Here's the code we had before, only now it's in the body of the init function.

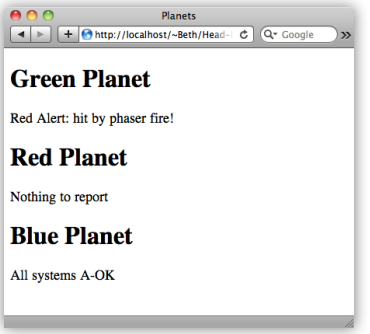
```
<script>
function init() {
    var planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
}

window.onload = init;
</script>
```

Here, we're assigning the function init to the window.onload property. Make sure you don't use parentheses after the function name! We're not calling the function; we're just assigning the function value to the window.onload property.

Let's try that again...

Go ahead and reload the page with the new `init` function and the `onload` property. This time the browser will load the page completely, build the entire DOM and *only then* call your `init` function.

Ah, there we go, now the green planet shows the Red Alert, just like we wanted.

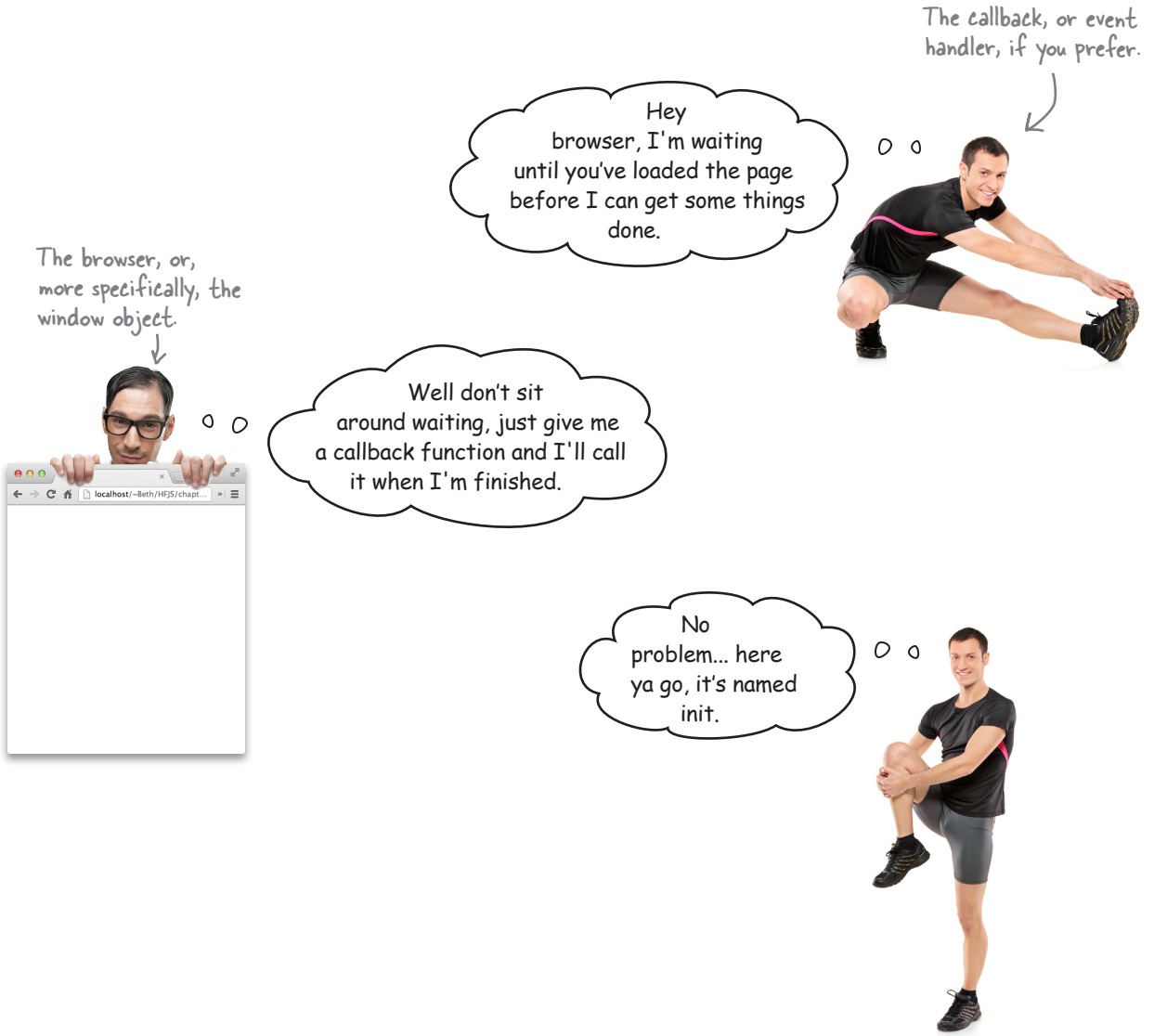


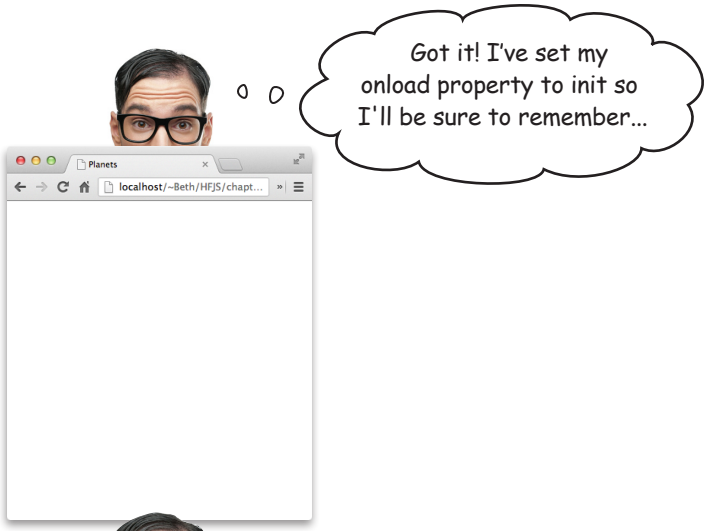
You say "event hander," I say "callback"

Let's think about how `onload` works just a bit more, because it uses a common coding pattern you'll see over and over again in JavaScript.

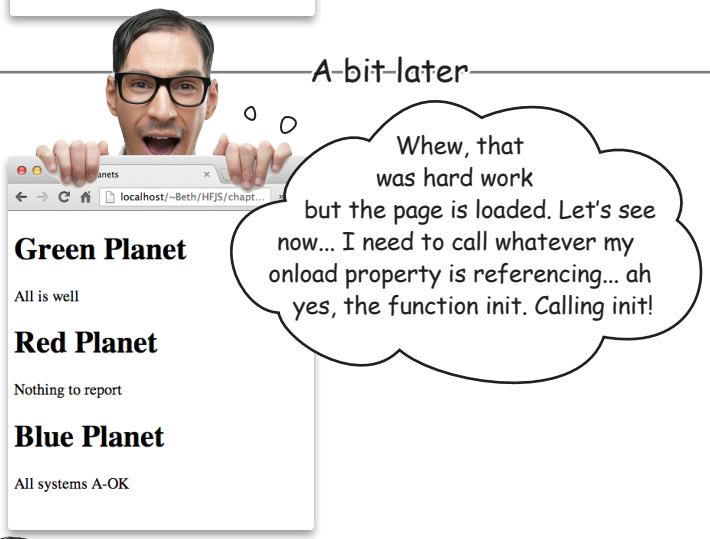
Let's say there's a big important event that's going to occur, and you *definitely* want to know about it. Say that event is the "page is loaded" event. Well, a common way to deal with that situation is through a *callback*, also known as an *event handler.*

A callback works like this: give a function to the object that knows about the event. When the event occurs, that object will call you back, or notify you, by calling that function. You're going to see this pattern in JavaScript for a variety of events.

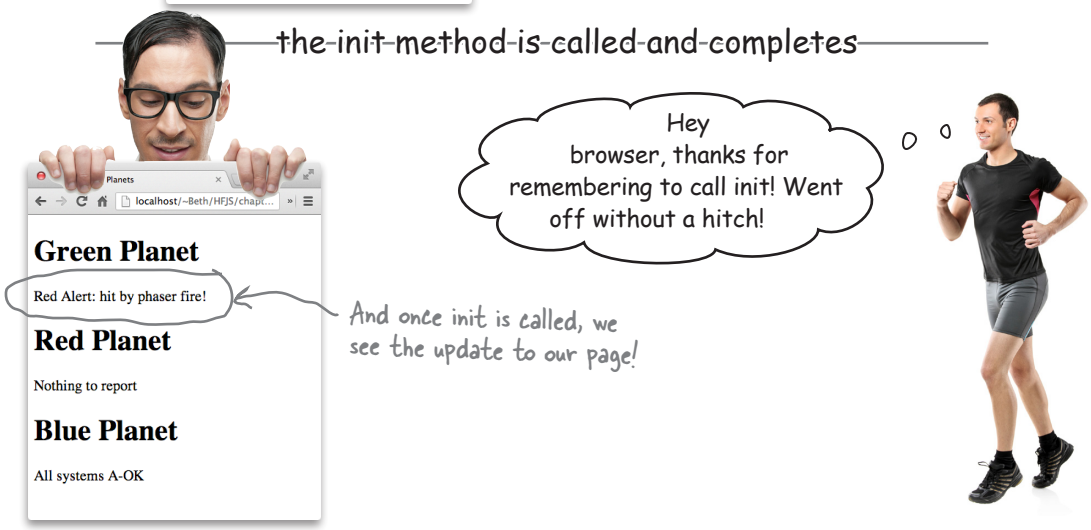


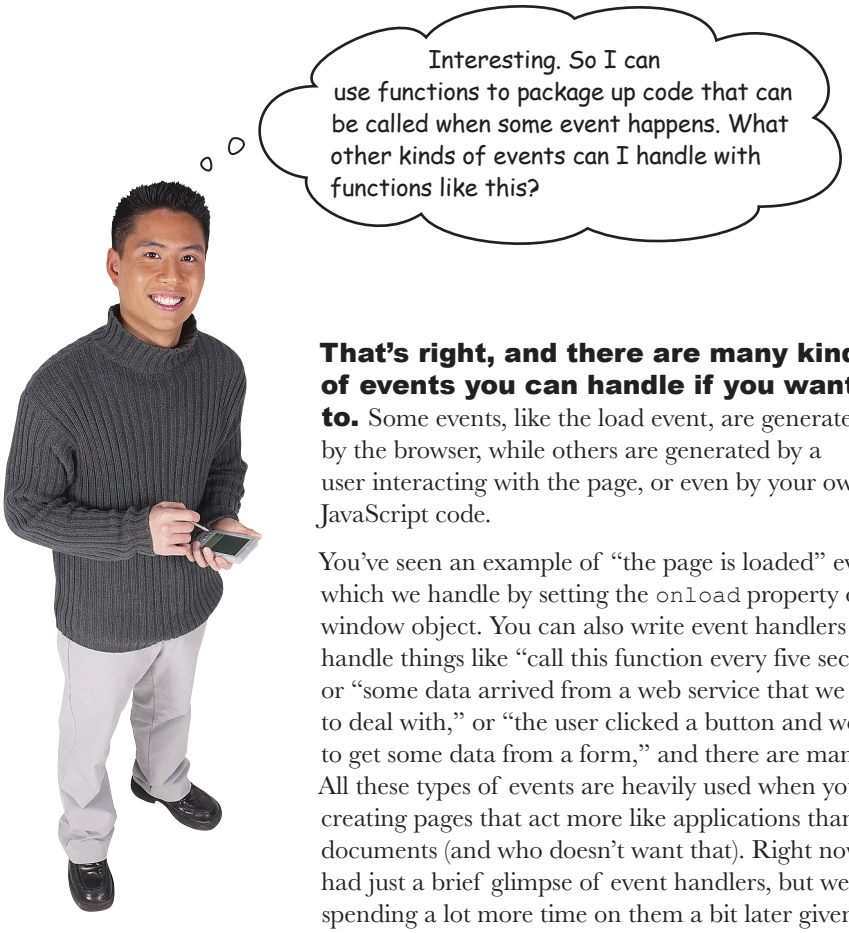


A bit later



the init method is called and completes





That's right, and there are many kinds of events you can handle if you want to. Some events, like the load event, are generated by the browser, while others are generated by a user interacting with the page, or even by your own JavaScript code.

You've seen an example of "the page is loaded" event, which we handle by setting the `onload` property of the window object. You can also write event handlers that handle things like "call this function every five seconds," or "some data arrived from a web service that we need to deal with," or "the user clicked a button and we need to get some data from a form," and there are many more. All these types of events are heavily used when you're creating pages that act more like applications than static documents (and who doesn't want that). Right now, we've had just a brief glimpse of event handlers, but we'll be spending a lot more time on them a bit later given their important role in JavaScript programming.

Sharpen your pencil

Here's some HTML for a playlist of songs, except that the list is empty. It's your job to complete the JavaScript below to add the songs to the list. Fill in the blank with the JavaScript that will do the job. Check your answer with our solution at the end of the chapter before you go on.

Here's the HTML for the page.

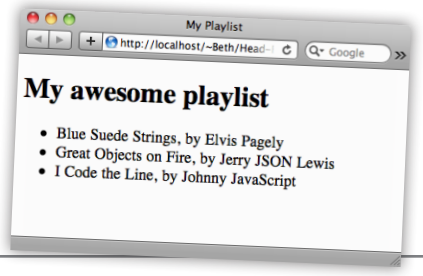
```
<!doctype html>
<html lang="en">
<head>
  <title>My Playlist</title>
  <meta charset="utf-8">
  <script>

  _____ addSongs() {
    var song1 = document._____("_____");
    var _____ = _____("_____");
    var _____ = _____.getElementById("_____");


    _____.innerHTML = "Blue Suede Strings, by Elvis Pagely";
    _____ = "Great Objects on Fire, by Jerry JSON Lewis";
    song3._____ = "I Code the Line, by Johnny JavaScript";
  }
  window._____ = _____;
  </script>
</head>
<body>
  <h1>My awesome playlist</h1>
  <ul id="playlist">
    <li id="song1"></li>
    <li id="song2"></li>
    <li id="song3"></li>
  </ul>
</body>
</html>
```

Here's our script. This code should fill in the list of songs below, in the <ul>.

Fill in the blanks with the missing code to get the playlist filled out.

Here's the empty list of songs. The code above should add content to each <li> in the playlist.

When you get the JavaScript working, this is what the web page will look like after you load the page.



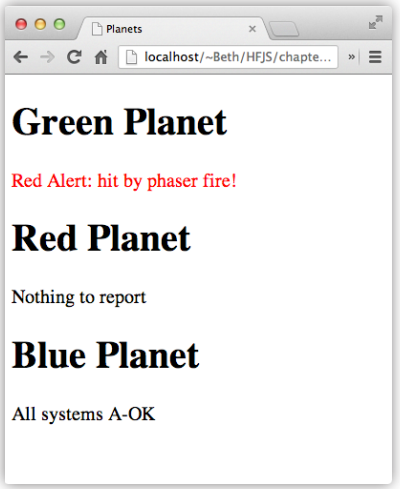
Why stop now? Let's take it further

Let's think for a second about what you just did: you took a static web page and you dynamically changed the content of one of its paragraphs *using code*. It seems like a simple step, but this is really the beginning of making a *truly interactive* page.

And that's our goal, which we'll fully realize in Chapter 8.

Let's take the second step: now that you know how to get your hands on an element in the DOM, let's set an *attribute* of an element with code.

Why would that be interesting? Well, take our simple planets example. When we set the paragraph to read "Red Alert," we could also set the paragraph's color to red. That would certainly more clearly communicate our message.



Here's how we're going to do that:

1. We'll define a CSS rule for the class "redtext" that specifies a red color for the text of the paragraph. That way any element we add to this class will have red text.
2. Next, we'll add code to take the greenplanet paragraph element and add the class "redtext".

That's it. All we need now is to learn how to set an attribute of an element and then we can write the code.

Sharpen your pencil

How about getting another part of your brain working? We're going to need the CSS style for the class "redtext" that sets the color to "red" for the text in the planet paragraph. If it's been a while since you wrote CSS, don't worry; give it a shot anyway. If you can do it in your sleep, awesome. Either way, you'll find the answer at the end of this chapter.

How to set an attribute with setAttribute

Element objects have a method named `setAttribute` that you can call to set the value of an HTML element's attribute. The `setAttribute` method looks like this:

We take our element object.

```
planet.setAttribute("class", "redtext");
```

Note if the attribute doesn't exist a new one will be created in the element.

And we use its setAttribute method to either add a new attribute or change an existing attribute.

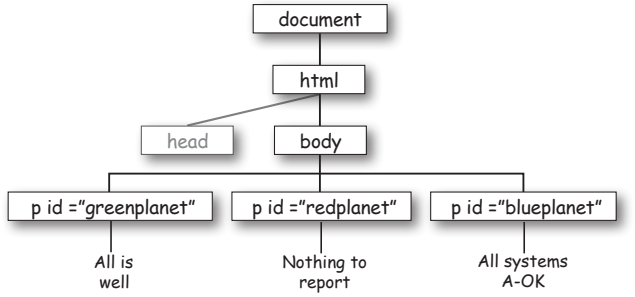
The method takes two arguments, the name of the attribute you want to set or change...

... and the value you'd like to set that attribute to.

We can call `setAttribute` on any element to change the value of an existing attribute, or, if the attribute doesn't already exist, to add a new attribute to the element. As an example, let's check out how executing the code above affects our DOM.

Before...

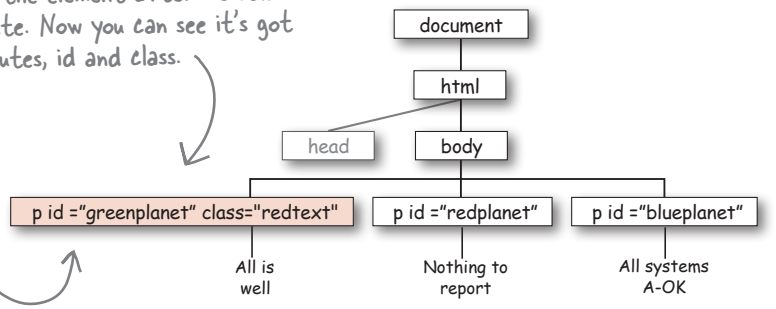
Here's the element before we call the setAttribute method on it. Notice this element already has one attribute, id.



And After

And here's the element after we call setAttribute. Now you can see it's got two attributes, id and class.

Remember, when we call the setAttribute method, we're changing the element object in the DOM, which immediately changes what you see displayed in the browser.



More fun with attributes!
(you can GET attributes too)

Need to know the value of an attribute in an element? No problem, we have a `getAttribute` method that you can call to get the value of an HTML element's attribute.

Get a reference to the element with getElementById, then use the element's getAttribute method to get the attribute.

Pass in the name of the attribute you want the value of.

```
var scoop = document.getElementById("raspberry");
var altText = scoop.getAttribute("alt");
console.log("I can't see the image in the console,");
console.log(" but I'm told it looks like: " + altText);
```

What happens if my attribute doesn't exist in the element?

Remember what happens when you call `getElementById` and the id doesn't exist in the DOM? You get null. Same thing with `getAttribute`. If the attribute doesn't exist, you'll get back null. Here's how you test for that:

```
var scoop = document.getElementById("raspberry");
var altText = scoop.getAttribute("alt");
if (altText == null) {
     console.log("Oh, I guess there isn't an alt attribute.");
} else {
     console.log("I can't see the image in the console,");
     console.log(" but I'm told it looks like " + altText);
}
```

Test to make sure there actually is an attribute value returned.

If there's no attribute value, we do this...

... and if there is one, we can show the text content in the console.

Don't forget getElementById can return null too!

Any time you ask for something, you need to make sure you got back what you expected...

The call to getElementById can return a null value if the element id does not exist in the DOM. So, to follow best practices, you'll want to make sure you test for null after getting elements too. We could follow that rule ourselves, but then the book would end up being 1000 pages longer.

Meanwhile, back at the ~~ranch~~ solar system...

It's time to put all the code for the planets together and do a final test drive.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planets</title>
  <style>
    .redtext { color: red; }
  </style>
  <script>
    function init() {
        var planet = document.getElementById("greenplanet");
        planet.innerHTML = "Red Alert: hit by phaser fire!";
        planet.setAttribute("class", "redtext");
    }
    window.onload = init;
  </script>
</head>
<body>
  <h1>Green Planet</h1>
  <p id="greenplanet">All is well</p>
  <h1>Red Planet</h1>
  <p id="redplanet">Nothing to report</p>
  <h1>Blue Planet</h1>
  <p id="blueplanet">All systems A-OK</p>
</body>
</html>
```

Here's all the HTML, CSS and JavaScript for the planets.

We've got the redtext class included here so when we add "redtext" as the value for the class attribute in our code, it turns the text red.

And to review: we're getting the greenplanet element, and stashing the value in the planet variable. Then we're changing the content of the element, and finally adding a class attribute that will turn the text of the element red.

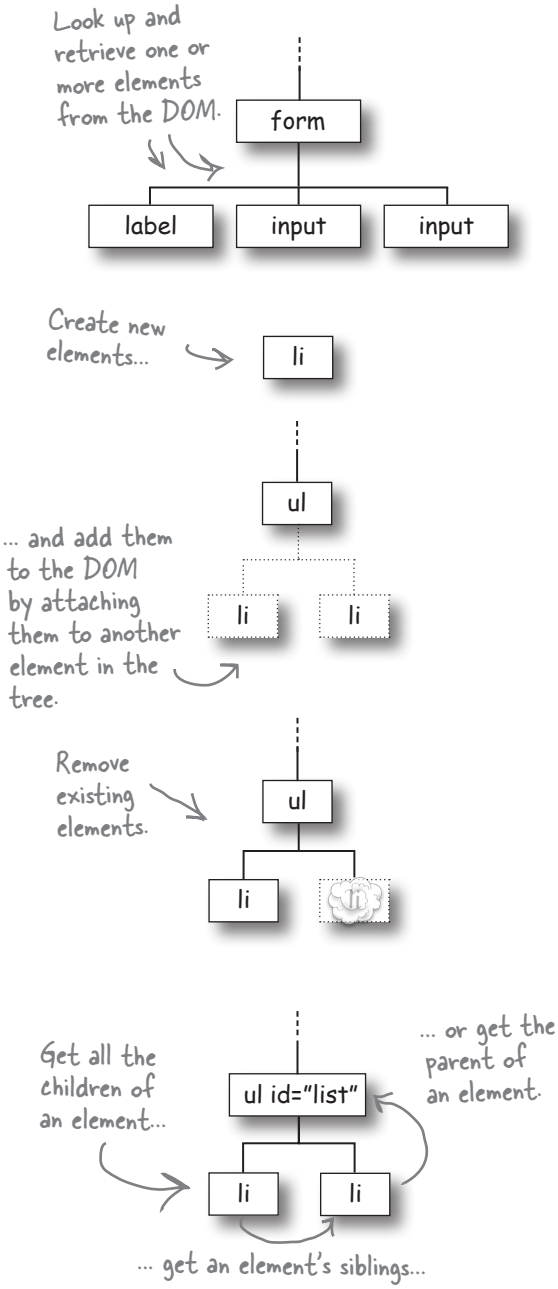
We're calling the init function only when the page is fully loaded!

Test driving the planets one last time...

Load this page up in your browser and you'll see the green planet has been hit by phaser fire, and now we see the message in bright red, so we'll be sure not to miss it!

So what else is a DOM good for anyway?

The DOM can do a fair bit more than we've seen so far and we'll be seeing some of its other functionality as we move forward in the book, but for now let's just take a quick look so you've got it in the back of your mind:



Get elements from the DOM.

Of course you already know this because we've been using document.getElementById, but there are other ways to get elements as well; in fact, you can use tag names, class names and attributes to retrieve not just one element, but a whole set of elements (say all elements in the class "on_sale"). And you can get form values the user has typed in, like the text of an input element.

Create and add elements to the DOM.

You can create new elements and you can also add those elements to the DOM. Of course, any changes you make to the DOM will show up immediately as the DOM is rendered by the browser (which is a good thing!).

Remove elements from the DOM.

You can also remove elements from the DOM by taking a parent element and removing any of its children. Again, you'll see the element removed in your browser window as soon as it is deleted from the DOM.

Traverse the elements in the DOM.

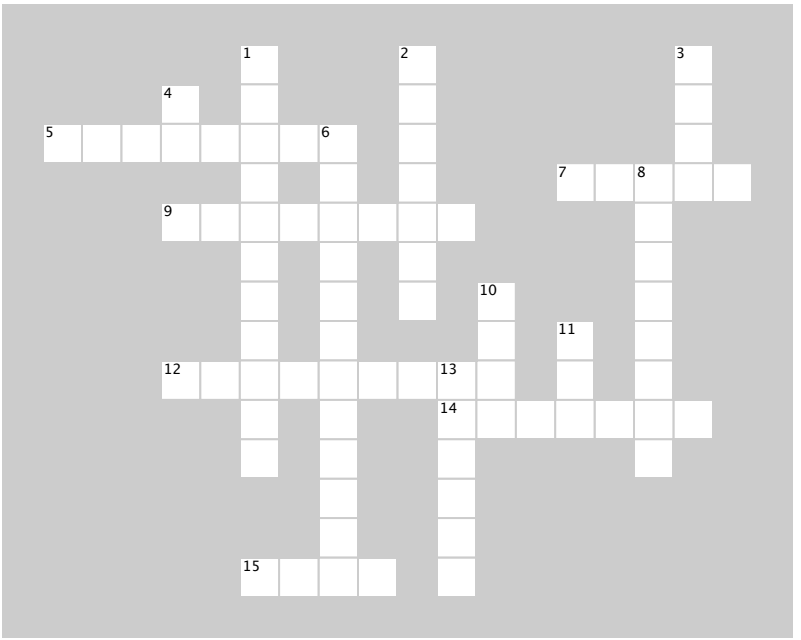
Once you have a handle to an element, you can find all its children, you can get its siblings (all the elements at the same level), and you can get its parent. The DOM is structured just like a family tree!

BULLET POINTS

- The **Document Object Model**, or DOM, is the browser's internal representation of your web page.
- The browser creates the DOM for your page as it loads and parses the HTML.
- You get access to the DOM in your JavaScript code with the document object.
- The document object has properties and methods you can use to access and modify the DOM.
- The **document.getElementById** method grabs an element from the DOM using its id.
- The document.getElementById method returns an **element object** that represents an element in your page.
- An element object has properties and methods you can use to read an element's content, and change it.
- The **innerHTML** property holds the text content, as well as all nested HTML content, of an element.
- You can modify the content of an element by changing the value of its innerHTML property.
- When you modify an element by changing its innerHTML property, you see the change in your web page immediately.
- You can get the value of an element's attributes using the **getAttribute** method.
- You can set the value of an element's attributes using the **setAttribute** method.
- If you put your code in a <script> element in the <head> of your page, you need to make sure you don't try to modify the DOM until the page is fully loaded.
- You can use the window object's **onload** property to set an **event handler**, or callback, function for the load event.
- The event handler for the window's onload property will be called as soon as the page is fully loaded.
- There are many different kinds of events we can handle in JavaScript with event handler functions.

JavaScript cross

Load the DOM into your brain with this puzzle.



ACROSS

5. Functions that handle events are known as event _____.

7. Dr. Evel's passcode clue was in the element with the id _____.

9. Assign a _____ to the window.onload property to handle the load event.

12. Use the element object's property, _____, to change the HTML inside an element.

14. The setAttribute method is a method of an _____ object.

15. The DOM is shaped like a _____.

DOWN

1. Which planet gets hit by phaser fire?

2. Use the _____ to see if you have errors in your code.

3. It's important to make sure the ____ is completely loaded before using code to get or change elements in the page.

4. The getElementById method gets an element by its ___.

6. Change the class of an element using the _____ method.

8. The _____ object is always at the top of the DOM tree.

10. It's a good idea to check for ____ when using getElementById.

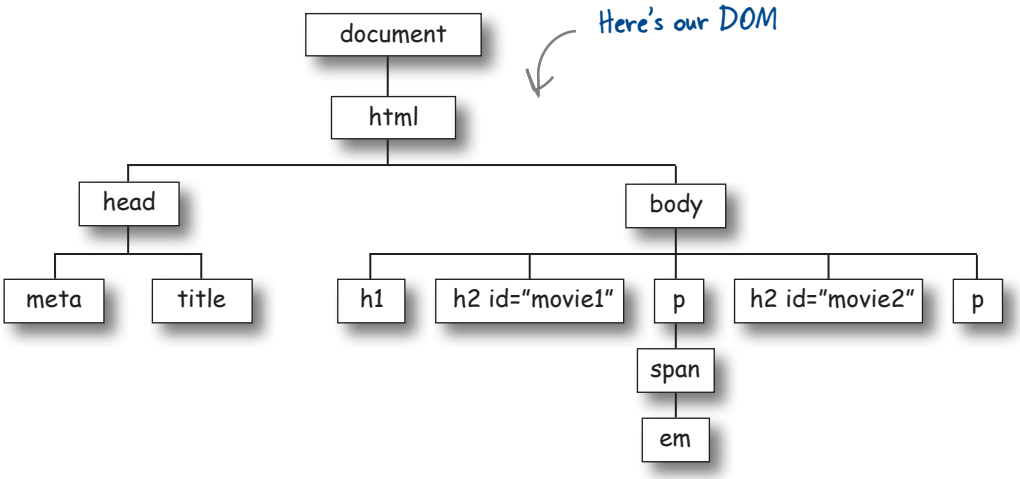
11. When you load a page into the browser, the browser creates a ____ representing all the elements and content in the page.

13. getElementById is a _____ of the document object.

BE the Browser Solution

Your job is to act like you're the browser. You need to parse the HTML and build your very own DOM from it. Go ahead and parse the HTML to the right, and draw your DOM below. We've already started it for you.


```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Movies</title>
  </head>
  <body>
    <h1>Movie Showtimes</h1>
    <h2 id="movie1" >Plan 9 from Outer Space</h2>
    <p>Playing at 3:00pm, 7:00pm.
      <span>
        Special showing tonight at <em>midnight</em>!
      </span>
    </p>
    <h2 id="movie2">Forbidden Planet</h2>
    <p>Playing at 5:00pm, 9:00pm.</p>
  </body>
</html>
```



Sharpen your pencil Solution

Here's some HTML for a playlist of songs, except that the list is empty. It's your job to complete the JavaScript below to add the songs to the list. Fill in the blank with the JavaScript that will do the job. Here's our solution.

Here's the HTML for the page.

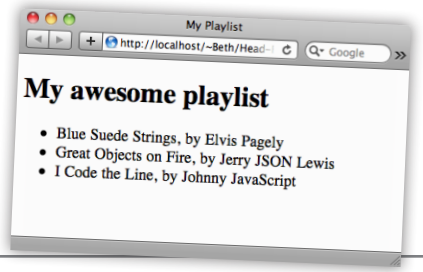
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My Playlist</title>
  <script>
    function addSongs() {
      var song1 = document.getElementById("song1");
      var song2 = document.getElementById("song2");
      var song3 = document.getElementById("song3");

      song1.innerHTML = "Blue Suede Strings, by Elvis Pagely";
      song2.innerHTML = "Great Objects on Fire, by Jerry JSON Lewis";
      song3.innerHTML = "I Code the Line, by Johnny JavaScript";
    }
    window.onload = addSongs;
  </script>
</head>
<body>
  <h1>My awesome playlist</h1>
  <ul id="playlist">
    <li id="song1"></li>
    <li id="song2"></li>
    <li id="song3"></li>
  </ul>
</body>
</html>
```

Here's our script. This code should fill in the list of songs below, in the <ul>.

Fill in the blanks with the missing code to get the playlist filled out.

Here's the empty list of songs. The code above should add content to each <li> in the playlist.

This is what the web page looks like after you load the page.



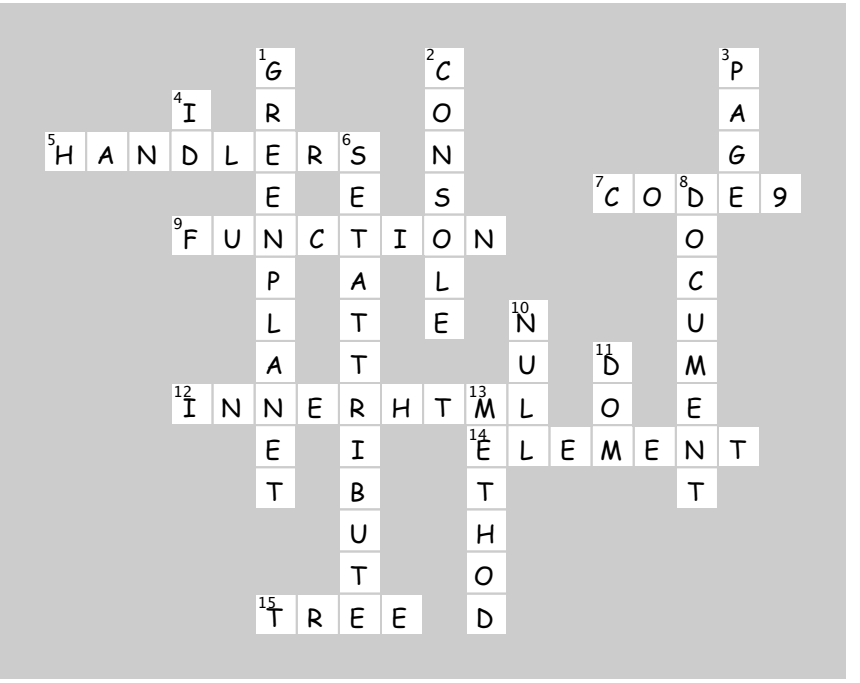
Sharpen your pencil Solution

How about getting another part of your brain working? We're going to need the CSS style for the class "redtext" that sets the color to "red" for the text in the planet paragraph. If it's been a while since you wrote CSS, don't worry; give it a shot anyway. If you can do it in your sleep, awesome. Here's our solution.

```
.redtext { color: red; }
```

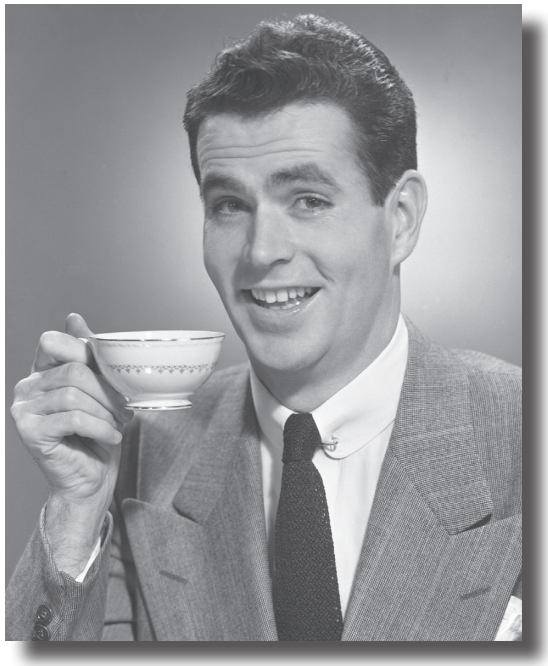
JavaScript cross Solution

Load the DOM into your brain with this puzzle.



7 types, equality, conversion and all that jazz

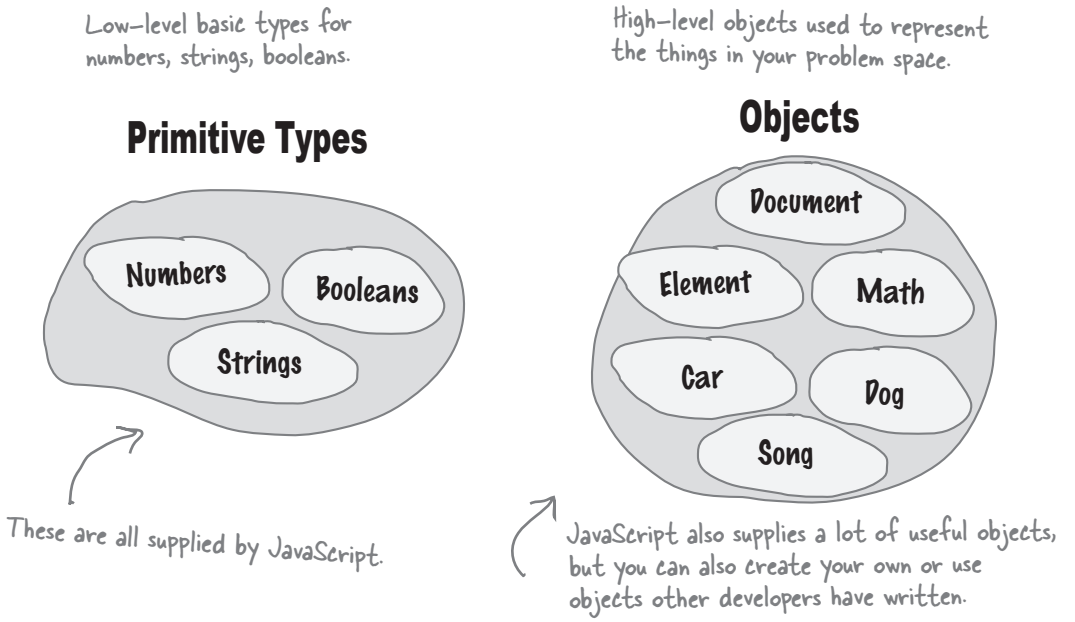
Serious types



It's time to get serious about our types. One of the great things about JavaScript is you can get a long way without knowing a lot of details of the language. But to truly **master the language**, get that promotion and get on to the things you really want to do in life, you have to rock at **types**. Remember what we said way back about JavaScript? That it didn't have the luxury of a silver-spoon, academic, peer-reviewed language definition? Well that's true, but the academic life didn't stop Steve Jobs and Bill Gates, and it didn't stop JavaScript either. It does mean that JavaScript doesn't have the... well, the most thought-out type system, and we'll find a few **idiosyncrasies** along the way. But, don't worry, in this chapter we're going to nail all that down, and soon you'll be able to avoid all those embarrassing moments with types.

The truth is out there...

Now that you've had a lot of experience working with JavaScript types—there's your primitives with numbers, strings, and booleans, and there's all the objects, some supplied by JavaScript (like the Math object), some supplied by the browser (like the document object), and some you've written yourself—aren't you just basking in the glow of JavaScript's simple, powerful and consistent type system?



After all what else would you expect from the official language of Webville? In fact, if you were a mere scripter, you might think about sitting back, sipping on that Webville Martini, and taking a much needed break…

But you're not a mere scripter, and something is amiss. You have that sinking feeling that behind Webville's picket fences something bizarre is at work. You've heard the reports of sightings of strings that are acting like objects, you've read in the blogs about a (probably radioactive) null type, you've heard the rumors that the JavaScript interpreter as of late has been doing some weird type conversion. What does it all mean? We don't know, but the truth is out there and we're going to uncover it in this chapter, and when we do, we might just turn what you think of true and false upside down.

A bunch of JavaScript values and party crashers, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Draw an arrow from each sentence to the name of one attendee. We've already guessed one of them for you. Check your answers at the end of the chapter before you go on.

If you find this exercise difficult, it's okay to cheat and look at the answers.

I get returned from a function when there is no return statement.

I'm the value of a variable when I haven't been assigned a value.

I'm the value of an array item that doesn't exist in a sparse array.

I'm the value of a property that doesn't exist.

I'm the value of a property that's been deleted.

I'm the value that can't be assigned to a property when you create an object.

Tonight's attendees:

zero

empty object

null

undefined

NaN

infinity

area 51

...---...

{}

[]

Watch out, you might bump into undefined when you aren't expecting it...

As you can see, whenever things get shaky—you need a variable that's not been initialized yet, you want a property that doesn't exist (or has been deleted), you go after an array item that isn't there—you're going to encounter `undefined`.

But what the heck is it? It's not really that complicated. Think of `undefined` as the value assigned to things that don't yet have a value (in other words they haven't been initialized).

So what good is it? Well, `undefined` gives you a way to test to see if a variable (or property, or array item) has been given a value. Let's look at a couple of examples, starting with an unassigned variable:

You can check to see if a variable like x is undefined. Just compare it to the value undefined.

```
var x;

if (x == undefined) {
    // x isn't defined! just deal with it!
}
```

Note that we're using the value undefined here, not to be confused with the string "undefined".

Or, how about an object property:

You can check to see if a property is undefined, again by comparing it to the value undefined.

```
var customer = {
    name: "Jenny"
};
if (customer.phoneNumber == undefined) {
    // get the customer's phone number
}
```

there are no Dumb Questions

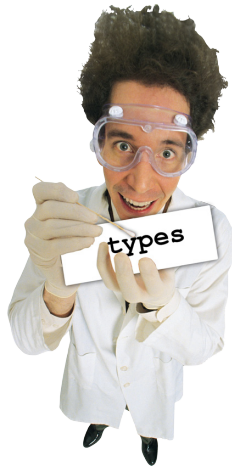
Q: When do I need to check if a variable (or property or array item) is undefined?

A: Your code design will dictate this. If you've written code so that a property or variable may not have a value when a certain block of code is executed, then checking for undefined gives you a way to handle that situation rather than computing with undefined values.

Q: If undefined is a value, does it have a type?

A: Yes, it does. The type of undefined is undefined. Why? Well our logic (work with us here) is this: it isn't an object, or a number or a string or a boolean, or really anything that is defined. So why not make the type undefined, too? This is one of those weird twilight zones of JavaScript you just have to accept.

IN THE LABORATORY



In the laboratory we like to take things apart, look under the hood, poke and prod, hook up our diagnostic tools and check out what is really going on. Today, we're investigating JavaScript's type system and we've found a little diagnostic tool called **typeof** to examine variables. Put your lab coat and safety goggles on, and come on in and join us.

The **typeof** operator is built into JavaScript. You can use it to probe the type of its operand (the thing you use it to operate on). Here's an example:

```
var subject = "Just a string";

var probe = typeof subject;
console.log(probe);
```

The typeof operator takes an operand, and evaluates to the type of the operand.

The type here is "string". Note that typeof uses strings to represent types, like "string", "boolean", "number", "object", "undefined", and so on.

JavaScript console

```
string
```

Now it's your turn. Collect the data for the following experiments:

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var test8 = ["abcdef", 123];
function test9(){return "abcdef"};

console.log(typeof test1);
console.log(typeof test2);
console.log(typeof test3);
console.log(typeof test4);
console.log(typeof test5);
console.log(typeof test6);
console.log(typeof test7);
console.log(typeof test8);
console.log(typeof test9);
```

Here's the test data, and the tests.

JavaScript console

Put your results here. Are there any surprises?

I remember from the DOM chapter that getElementById returns null, not undefined, if the id doesn't exist. What exactly is null, and why doesn't getElementById return undefined instead?

Ah yes, this causes a lot of confusion. There are many languages that have the concept of a value that means "no object." And, it's not a bad idea—take the `document.getElementById` method. It's supposed to return an object right? So, what happens if it can't? Then we want to return something that says "I would have been an object if there was one, but we don't have one." And that's what `null` is.

You can also set a variable to `null` directly:

```
var killerObjectSomeday = null;
```

What does it mean to assign the value `null` to a variable? How about "We intend to assign an object to this variable at some point, but we haven't yet."

Now, if you're scratching your head and saying "Hmm, why didn't they just use `undefined` for that?" then you're in good company. The answer comes from the very beginnings of JavaScript. The idea was to have one value for variables that haven't been initialized to anything yet, and another that means the lack of an object. It isn't pretty, and it's a little redundant, but it is what it is at this point. Just remember the intent of each (`undefined` and `null`), and know that it is most common to use `null` in places where an object should be but one can't be created or found, and it is most common to find `undefined` when you have a variable that hasn't been initialized, or an object with a missing property, or an array with a missing value.

BACK IN THE LABORATORY

Oops, we forgot null in our test data. Here's the missing test case:

```
var test10 = null;

console.log(typeof test10);
```

Put your results here.

JavaScript console

How to use null

There are many functions and methods out there in the world that return objects, and you'll often want to make sure what you're getting back is a full-fledged object, and not `null`, just in case the function wasn't able to find one or make one to return to you. You've already seen examples from the DOM where a test is needed:

Let's look for the all-important header element.

```
var header = document.getElementById("header");
if (header == null) {
     // okay, something is seriously wrong if we have no header
}
```

Uh oh, it doesn't exist. Abandon ship!

Keep in mind that getting `null` doesn't necessarily mean something is wrong. It may just mean something doesn't exist yet and needs to be created, or something doesn't exist and you can skip it. Let's say users have the ability to open or close a weather widget on your site. If a user has it open there's a `<div>` with the id of "weatherDiv", and if not, there isn't. All of a sudden `null` becomes quite useful:

Let's see if the element with id "weatherDiv" exists.

```
var weather = document.getElementById("weatherDiv");
if (weather != null) {
     // create content for the weather div
}
```

If the result of getElementById isn't null, then there is such an element in the page. Let's create a nice weather widget for it (presumably getting the weather for the local area).

We can use null to check to see if an object exists yet or not.

Remember, null is intended to represent an object that isn't there.

WICKEDLYSMART'S
Believe It or Not!!

The Number that isn't a Number

It's easy to write JavaScript statements that result in numeric values that are not well defined.

Here are a few examples:

```
var a = 0/0;
```

In mathematics this has no direct answer, so we can't expect JavaScript to know the answer either!

```
var b = "food" * 1000;
```

We don't know what this evaluates to, but it is certainly not a number!

```
var c = Math.sqrt(-9);
```

If you remember high school math, the square root of a negative number is an imaginary number, which you can't represent in JavaScript.

Believe it or not, there are numeric values that are **impossible to represent** in JavaScript! JavaScript can't express these values, so it has a stand-in value that it uses:

NaN

JavaScript uses the value NaN, more commonly known as "Not a Number", to represent numeric results that, well, can't be represented. Take 0/0 for instance. 0/0 evaluates to something that just can't be represented in a computer, so it is represented by NaN in JavaScript.

NaN MAY BE THE WEIRDEST VALUE IN THE WORLD. Not only does it represent all the numeric values that can't be represented, it is the only value in JavaScript that isn't equal to itself!

You heard that right. If you compare NaN to NaN, they aren't equal!

NaN != NaN

Dealing with NaN

Now you might think that dealing with `NaN` is a rare event, but if you're working with any kind of code that uses numbers, you'd be surprised how often it shows up. The most common thing you'll need to do is test for `NaN`, and given everything you've learned about JavaScript, how to do this might seem obvious:

```
if (myNum == NaN) {
    myNum = 0;
}
```

You'd think this would work, but it doesn't.

WRONG!

Any sensible person would assume that's how you test to see if a variable holds a `NaN` value, but it doesn't work. Why? Well, `NaN` isn't equal to anything, not even itself, so, any kind of test for equality with `NaN` is off the table. Instead you need to use a special function: `isNaN`. Like this:

```
if (isNaN(myNum)) {
    myNum = 0;
}
```

Use the isNaN function, which returns true if the value passed to it is not a number.

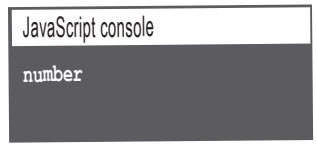
RIGHT!

It gets even weirder

So, let's think through this a bit more. If `NaN` stands for "Not a Number", what is it? Wouldn't it be easier if it were named for what it is rather than what it isn't? What do you think it is? We can check its type for a hint:

```
var test11 = 0 / 0;
console.log(typeof test11);
```

Here's what we got.



If your mind isn't blown, you should probably just use this book for some good kindling.

What on earth? `NaN` is of type number? How can something that's not a number have the type number? Okay, deep breath. Think of `NaN` as just a poorly named value. Someone should have called it something more like "Number that can't be represented" (okay, we agree the acronym isn't quite as nice) instead of "Not a Number". If you think about it like that, then you can think of `NaN` as being a value that is a number but can't be represented (at least, not by a computer).

Go ahead and add this one to your JavaScript twilight zone list.

there are no Dumb Questions

Q: If I pass isNaN a string, which isn't a number, will it return true?

A: It sure will, just as you'd expect. You can expect a variable holding the value NaN, or any other value that isn't an actual number to result in isNaN returning true (and false otherwise). There are a few caveats to this that you'll see when we talk about type conversion.

Q: But why isn't NaN equal to itself?

A: If you're deeply interested in this topic you'll want to seek out the IEEE floating point specification. However, the layman's insight into this is that just because NaN represents an unrepresentable numeric value, does not mean that those unrepresentable numbers are equal. For instance, take the sqrt(-1) and sqrt(-2). They are definitely not the same, but they both produce NaN.

Q: When we divide 0/0 we get NaN, but I tried dividing 10/0 and got `Infinity`. Is that different from `NaN`?

A: Good find. The Infinity (or -Infinity) value in JavaScript represents all numbers (to get a little technical) that exceed the upper limit on computer floating point numbers, which is 1.7976931348623157E+10308 (or -1.7976931348623157E+10308 for -Infinity). The type of Infinity is number and you can test for it if you suspect one of your values is getting a little large:

```
if (tamale == Infinity) {
    alert("That's a big tamale!");
}
```

Q: You did blow my mind with that "NaN is a number" thing. Any other mind blowing details?

A: Funny you should ask. How about Infinity minus Infinity equals.... wait for it..... NaN. We'll refer you to a good mathematician to understand that one.

Q: Just to cover every detail, did we say what the type of null is?

A: A quick way to find out is by using the typeof operator on null. If you do that you'll get back the result "object". And this makes sense from the perspective that null is used to represent an object that isn't there. However, this point has been heavily debated, and the most recent spec defines the type of null as null. You'll find this an area where your browser's JavaScript implementation may not match the spec, but, in practice, you'll rarely need to use the type of null in code.

We've been looking at some rather, um, interesting, values so far in this chapter. Now, let's take a look at some interesting behavior. Try adding the code below to the <script> element in a basic web page and see what you get in the console when you load up the page. You won't get why yet, but see if you can take a guess about what might be going on.

```
if (99 == "99") {
    console.log("A number equals a string!");
} else {
    console.log("No way a number equals a string");
}
```

Write what you get here.

JavaScript console

We have a confession to make

There is an aspect of JavaScript we've deliberately been holding back on. We could have told you up front, but it wouldn't have made as much sense as it will now.

It's not so much that we've been pulling the wool over your eyes, it's that there is more to the story than we've been telling you. And what is this topic? Here, let's take a look:

At some point a variable gets set, in this case to the number 99.

```
var testMe = 99;
```

And later it gets compared with a number in a conditional test.

```
if (testMe == 99) {
    // good things happen
}
```

Straightforward enough? Sure, what could be easier? However, one thing we've done at least once so far in this book, that you might not have noticed, is something like this:

At some point a variable gets set, in this case to the string "99".

Did we mention we're using a string this time?

```
var testMe = "99";
```

And later it gets compared with a number in a conditional test.

```
if (testMe == 99) {
    // good things happen
}
```

Now we have a string being compared to a number.

So what happens when we compare a number to a string? Mass chaos? Computer meltdown? Rioting in the streets?

No, JavaScript is smart enough to determine that 99 and "99" are the same for all practical purposes. But what exactly is going on behind the scenes to make this work? Let's take a look…

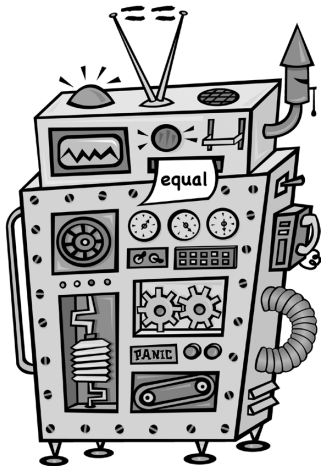
BULLET POINTS

Just a quick reminder about the difference between assignment and equality:

- `var x = 99;`
  = is the assignment operator. It is used to assign a value to a variable.

- `x == 99`
  == is a comparison operator. It is used to compare one value with another to see if they're equal.

Understanding the equality operator (otherwise known as ==)



You'd think that understanding equality would be a simple topic. After all, 1 == 1, "guacamole" == "guacamole" and true == true. But, clearly there is more at work here if "99" == 99. What could be going on inside the equality operator to make that happen?

It turns out the == operator takes the types of its operands (that is, the two things you're comparing) into account when it does a comparison. You can break this down into two cases:

If the two values have the same type, just compare them

If the two values you are comparing have the same type, like two numbers or two strings, then the comparison works just like you would expect: the two values are compared against each other and the result is true if they are the same value. Easy enough.

If the two values have different types, try to convert them into the same type and then compare them

This is the more interesting case. Say you have two values with different types that you want to compare, like a number and a string. What JavaScript does is convert the string into a number, and then compares the two values. Like this:

Note that the conversion is only temporary, so that the comparison can happen.

```
99 == "99"
```

When you're comparing a number and a string, JavaScript converts the string to a number (if possible)...

```
99 == 99
```

... and then tries the comparison again. Now, if they're equal, the expression results in true, false otherwise.

Okay, that makes some intuitive sense, but what are the rules here? What if I compare a boolean to a number, or null to undefined, or some other combination of values? How do I know what's going to get converted into what? And, why not convert the number into a string instead, or use some other scheme to test their equality? Well, this is defined by a fairly simple set of rules in the JavaScript specification that determine how the conversion happens when we compare two values with different types. This is one of those things you just need to internalize—once you've done that, you'll be on top of how comparisons work the rest of your JavaScript career.

This will also set you above your peers, and help you nail your next interview.

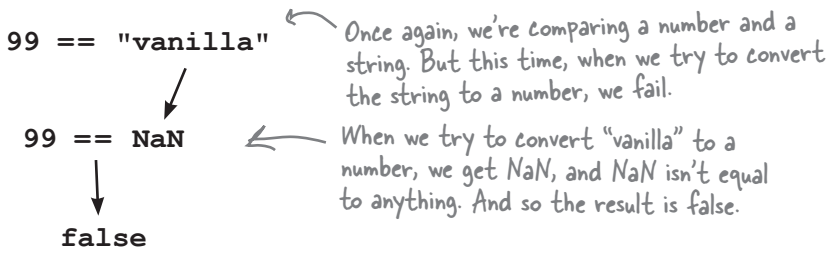
How equality converts its operands (sounds more dangerous than it actually is)

So what we know is that when you compare two values that have different types, JavaScript will convert one type into another in order to compare them. If you're coming from another language this might seem strange given this is typically something you'd have to code explicitly rather than have it happen automatically. But no worries, in general, it's a useful thing in JavaScript *so long as you understand when and how it happens*. And, that's what we've got to figure out now: when it happens and how it happens.

Here we go (in four simple cases):

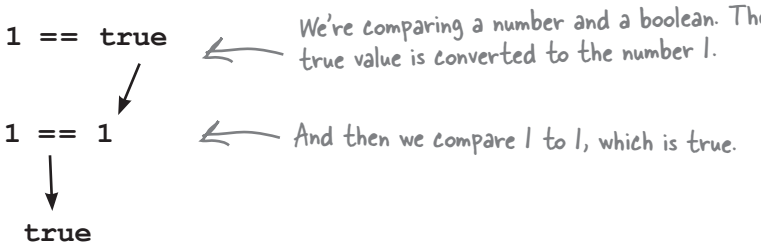
CASE#1: Comparing a number and a string.

If you're comparing a string and a number the same thing happens every time: the string is converted into a number, and the two numbers are then compared. This doesn't always go well, because not all strings can be converted to numbers. Let's see what happens in that case:

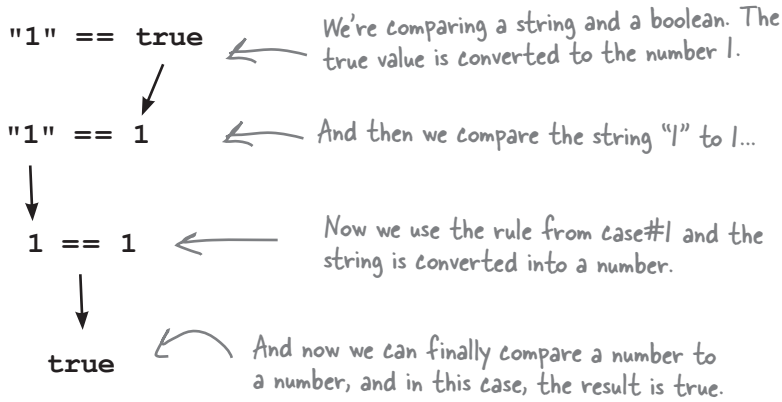


CASE#2: Comparing a boolean with any other type.

In this case, we convert the boolean to a number, and compare. This might seem a little strange, but it's easier to digest if you just remember that true converts to 1 and false converts to 0. You also need to understand that sometimes this case requires doing more than one type conversion. Let's look at a few examples:



Here's another case; this time a boolean is compared to a string. Notice how more steps are needed.

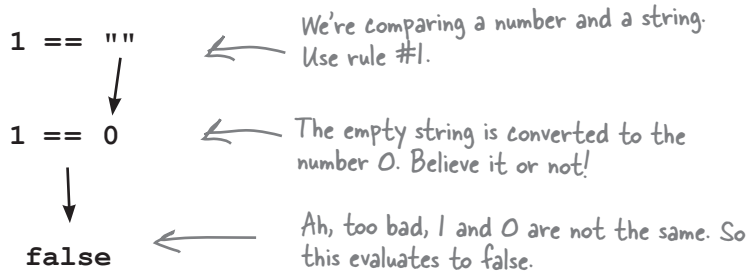


CASE#3: Comparing null and undefined.

Comparing these values evalutates to true. That might seem odd as well, but it's the rule. For some insight, these values both essentially represent "no value" (that is, a variable with no value, or an object with no value), so they are considered to be equal.

CASE#4: Oh, actually there is no case #4.

That's it. You can pretty much determine the value of any equality with these rules. That said, there are a few edge cases and caveats. One caveat is that we still need to talk about comparing objects, which we'll talk about in a bit. The other is around conversions that might catch you off guard. Here's one example:



If only I could find a way to test two values for **equality** without having to worry about their types being converted. A way to just test if two values are equal only if they have the same value *and* the same type. A way to not have to worry about all these rules and the mistakes they might cause. That would be dreamy. But I know it's just a fantasy...

How to get strict with equality

Editor's note: Make sure we have a photo release on file from Doug Crockford.

While we're making confessions, here's another one: there are not one, but *two equality operators*. You've already been introduced to == (equality), and the other operator is === (strict equality).

That's right, three equals. You can use === in place of == anytime you want, but before you start doing that, let's make sure you understand how they differ.

With ==, you now know all the complex rules around how the operands are converted (if they're different types) when they're compared. With ===, the rules are even more complicated.

Just kidding, actually there is *only one rule* with ===:

Two values are strictly equal only if they have the same type and the same value.

Read that again. What that means is, if two values have the same type we compare them. If they don't, forget it, we're calling it false no matter what—no conversion, no figuring out complex rules, none of that. All you need to remember is that === will find two values equal *only if they are the same type and the same value*.

Sharpen your pencil

For each comparison below write true or false below the operators == and === to represent the result of the comparison:

	==	===	
`"42" == 42`	true		`"42" === 42`
`"0" == 0`			`"0" === 0`
`"0" == false`			`"0" === false`
`"true" == true`			`"true" === true`
`true == (1 == "1")`			`true === (1 === "1")`

Tricky!

Tricky!

there are no Dumb Questions

Q: What happens if I compare a number, like 99, to a string, like "ninety-nine", that can't be converted to a number?

A: JavaScript will try to convert "ninety-nine" to a number, and it will fail, resulting in NaN. So the two values won't be equal, and the result will be false.

Q: How does JavaScript convert strings to numbers?

A: It uses an algorithm to parse the individual characters of a string and try to turn each one of them into a number. So if you write "34", it will look at "3", and see that can be a 3, and then it will look at "4" and see that can be a 4. You can also convert strings like "1.2" to floating point numbers—JavaScript is smart enough to recognize a string like this can still be a number.

Q: So, what if I try something like "true" == true?

A: That is comparing a string and a boolean, so according to the rules, JavaScript will first convert true to 1, and then compare "true" and 1. It will then try to convert "true" to a number, and fail, so you'll get false.

Q: So if there is both a == and a === operator, does that mean we have <= and <==, and >= and >==?

A: No. There are no <== and >== operators. You can use only <= and >=. These operators only know how to compare strings and numbers (true <= false doesn't really make sense), so if you try to compare any values other than two strings or two numbers (or a string and a number), JavaScript will attempt to convert the types using the rules we've discussed.

Q: So if I write 99 <= "100" what happens?

A: Use the rules: "100" is converted to a number, and then compared with 99. Because 99 is less than or equal to 100 (it's less than), the result is true.

Q: Is there a !==?

A: Yes, and just like === is stricter than ==, !== is stricter than !=. You use the same rules for !== as you do for ===, except that you're checking for inequality instead of equality.

Q: Do we use the same rules when we're comparing say, a boolean and a number with < and >, like 0 < true?

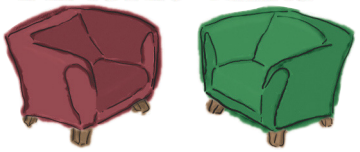
A: Yup! And in that case, true gets converted to 1, so you'll get true because 0 is less than 1.

Q: It makes sense for a string to be equal to another string, but how can a string be less than or greater than another string?

A: Good question. What does it mean to say "banana" < "mango"? Well, with strings, you can use alphabetical order to know if one string is less than or greater than another. Because "banana" begins with a "b" and "mango" with an "m", "banana" is less than "mango" because "b" comes before "m" in the alphabet. And "mango" is less than "melon" because, while the first letters are the same, when we compare the second letters, "a" comes before "e".

This alphabetical comparison can trip you up, however; for instance, "Mango" < "mango" is true, even though you might think that "M" is greater than "m" because its "M" is capitalized. The ordering of strings has to do with the ordering of the Unicode values that are used to represent each character in the computer (Unicode is a standard for representing characters digitally), and that ordering might not always be what you expect! For all the details, try googling "Unicode". But most of the time, the basic alphabetical ordering is all you need to know if one string is less than or greater than another.

Fireside Chats



Tonight's talk: **The equality and strict equality operators let us know who is boss.**

==

Ah look who it is, Mr. Uptight.

I'm up for a count of == versus === across all JavaScript code out in the world. You're going to come in way behind. It won't even be close.

I don't think so. I provide a valuable service. Who doesn't want to, say, compare user input in the form of a string to a number every once in a while?

When you were in grade school did you have to walk to school in the snow, every day, uphill, in both directions? Do you always have to do things the hard way?

The thing is, not only can I do the same comparisons you do, I add value on top of that by doing some nice conversions of types.

You'd rather just throw your hands up, call it false and go home?

===

Just keep in mind that several leading JavaScript gurus say that developers should use me, and only me. They think you should be taken out of the language altogether.

You know, you might be right, but folks are slowly starting to get it, and those numbers are changing.

And with it come all the rules you have to keep in mind to even use ==. Keep life and code simple; use === and if you need to convert user input to a number there are methods for that.

Very funny. There's nothing wrong with being strict and having clear-cut semantics around your comparisons. Bad, unexpected things can happen if you don't keep all the rules in mind.

Every time I look at your rules I throw up in my mouth a little. I mean comparing a boolean to anything means I convert the boolean to a number? That doesn't seem very sensical to me.

==

It's working so far. Look at all the code out there, a lot written by mere… well, scripters.

You mean like taking a shower after one of these conversations with you?

Hmm. Well, ever considered just buying me out? I'd be happy to go spend my days on the beach, kicking back with a margarita in hand.

Arguing about == versus === gets old. I mean there are more interesting things to do in life.

Look, here's the thing you have to deal with: people aren't going to just stop using ==. Sometimes it's really convenient. And people can use it in an educated way, taking advantage of it when it makes sense. Like the user input example—why the heck not use ==?

My new attitude is if people want to use you, great. I'm still here when they need me, and by the way, I still get a check every month no matter what they do! There's enough legacy code with == in the world—I'm never going off payroll.

===

No, but one can get a little too lax around your complex rules.

That's fine but pages are getting more complex, more sophisticated. It's time to take on some best practices.

No, like sticking to ===. It makes your code clearer and removes the potential for weird edge cases in comparisons.

I didn't see that coming, I thought you'd defend your position as THE equality operator until the end. What gives?

I don't even know how to respond.

Well like I said, you never know when something is going to happen.

Deep breath. There's a lot of debate around this topic, and different experts will tell you different things. Here's our take: traditionally, coders have used mostly == (equality) because, well, there wasn't a great awareness of the two operators and their differences. Today, we're more educated and for most purposes === (strict equality) works just fine and is in some ways the safer route because you know what you're getting. With ==, of course, you also know what you're getting, but with all the conversions it's hard sometimes to think through all the possibilities.

Now, there are times when == provides some nice convenience (like when you're comparing numbers to strings) and of course you should feel free to use == in those cases, especially now that, unlike many JavaScript coders, you know exactly what == does. Now that we've talked about ===, you'll see us mostly shift gears in this book and predominantly use ===, but we won't get dogmatic about it if there's a case where == makes our life easier and doesn't introduce issues.

You'll also hear developers refer to === (strict equality) as the "identity" operator.

We had our descriptions for these operators all figured out, and then they got all mixed up. Can you help us figure out who does what? Be careful, we're not sure if each contender matches zero, one or more descriptions. We've already figured one out, which is marked below:

`=`

`==`

`===`

`====`

Compares values to see if they are equal. This is the considerate equality operator. He'll go to the trouble of trying to convert your types to see if you are really equal.

Compares values to see if they are equal. This guy won't even consider values that have different types.

Assigns a value to a variable.

Compares object references and returns true if they are the same and false otherwise.

Even more type conversions...

Conditional statements aren't the only place you're going to see type conversion. There are a few other operators that like to convert types when they get the chance. While these conversions are meant to be a convenience for you, the coder, and often they are, it's good to understand exactly where and when they might happen. Let's take a look.

Another look at concatenation, and addition

You've probably figured out that when you use the + operator with numbers you get *addition*, and when you use it with strings you get *concatenation*. But what happens when we mix the types of +'s operands? Let's find out.

If you try to add a number and a string, JavaScript converts the number to a string and concatenates the two. Kind of the opposite of what it does with equality:

```
var addi = 3 + "4";
```

When we have a string added to a number, we get concatenation, not addition.

The result variable is set to "34" (not 7).

```
var plusi = "4" + 3;
```

Same here... we get "43".

If you put the string first and then use the + operator with a number, the same thing happens: the number is converted to a string and the two are joined by concatenation.

What about the other arithmetic operators?

When it comes the other arithmetic operators—like multiplication, division and subtraction—JavaScript prefers to treat those as arithmetic operations, not string operations.

```
var multi = 3 * "4";
```

Here, JavaScript converts the string "4" to the number 4, and multiplies it by 3, resulting in 12.

```
var divi = 80 / "10";
```

Here the string "10" is converted to the number 10. Then 80 is divided by the number 10, resulting in 8.

```
var mini = "10" - 5;
```

With minus, the "10" is converted to the number 10, so we have 10 minus 5, which is 5.

there are no Dumb Questions

Q: Is + always interpreted as string concatenation when one of the operands is a string?

A: Yes. However, because + has what is called left-to-right associativity, if you have a situation like this:

```
var order = 1 + 2 + " pizzas";
```

you'll get "3 pizzas", not "12 pizzas" because, moving left to right, 1 is added to 2 first (and both are numbers), which results in 3. Next we add 3 and a string, so 3 is converted to a string and concatenated with "pizza". To make sure you get the results you want, you can always use parentheses to force an operator to be evaluated first:

```
var order = (1 + 2) + " pizzas";
```

ensures you'll get 3 pizzas, and

```
var order = 1 + (2 + " pizzas");
```

ensures you'll get 12 pizzas.

Q: Is that it? Or are there more conversions?

A: There are some other places where conversion happens. For instance, the unary operator - (to make a negative number) will turn -true into -1. And concatenating a boolean with a string will create a string (like true + " love" is "true love"). These cases are fairly rare, and we've personally never needed these in practice, but now you know they exist.

Q: So if I want JavaScript to convert a string into a number to add it to another number, how would I do that?

A: There's a function that does this named Number (yes, it has a uppercase N). Use it like this:

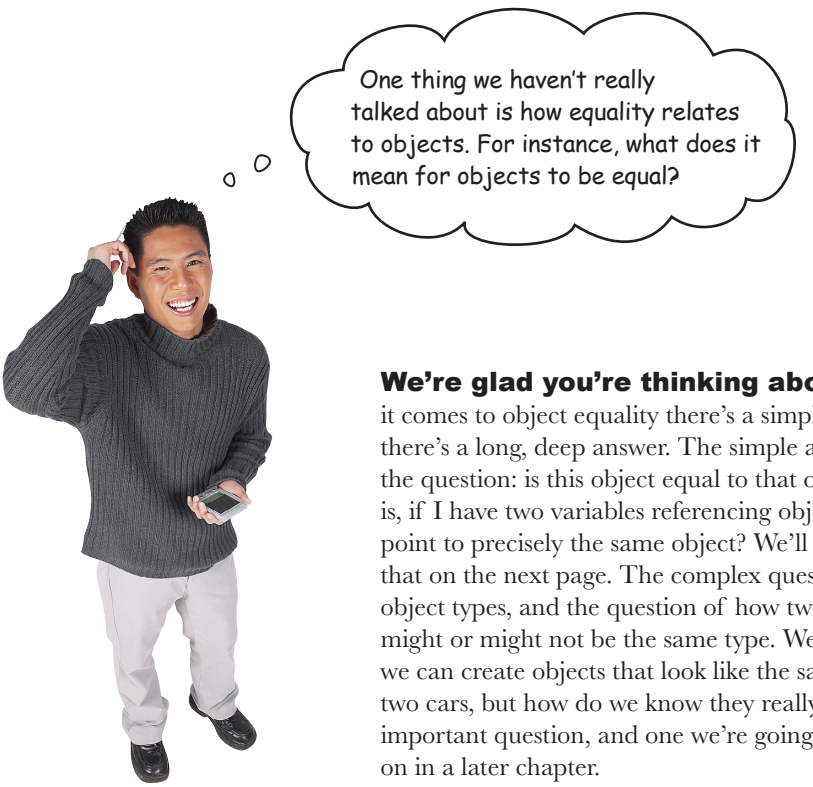
```
var num = 3 + Number("4");
```

This statement results in num being assigned the value 7. The Number function takes an argument, and if possible, creates a number from it. If the argument can't be converted to a number, Number returns.... wait for it..... NaN.

Sharpen your pencil

Time to test that conversion knowledge. For each expression below, write the result in the blank next to it. We've done one for you. Check your answers at the end of the chapter before you go on.

```
Infinity - "1"                    _____
"42" + 42                         "4242"
2 + "1 1"                         _____
99 + 101                          _____
"1" - "1"                         _____
console.log("Result: " + 10/2)    _____
3 + " bananas " + 2 + " apples"   _____
```



We're glad you're thinking about it. When it comes to object equality there's a simple answer and there's a long, deep answer. The simple answer tackles the question: is this object equal to that object? That is, if I have two variables referencing objects, do they point to precisely the same object? We'll walk through that on the next page. The complex question involves object types, and the question of how two objects might or might not be the same type. We've seen that we can create objects that look like the same type, say two cars, but how do we know they really are? It's an important question, and one we're going to tackle head on in a later chapter.

How to determine if two objects are equal

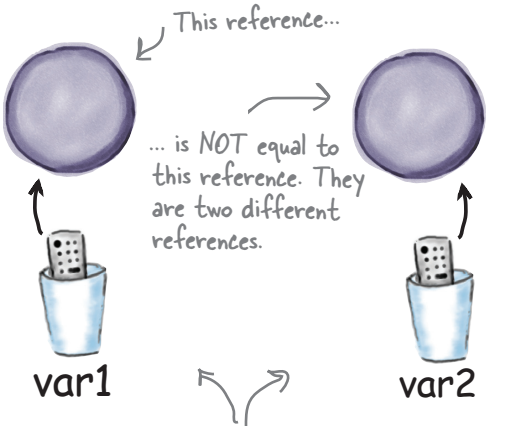
Your first question might be: are we talking about == or ===? Here's the good news: *if you're comparing two objects, it doesn't matter*! That is, if both operands are objects, then you can use either == or === because they work in exactly the same way. Here's what happens when you test two objects for equality:

When we test equality of two object variables, we compare the references to those objects

Remember, variables hold references to objects, and so whenever we compare two objects, we're comparing object references.

```
if (var1 === var2) {
    // wow, these are the same object!
}
```

Not in this case!

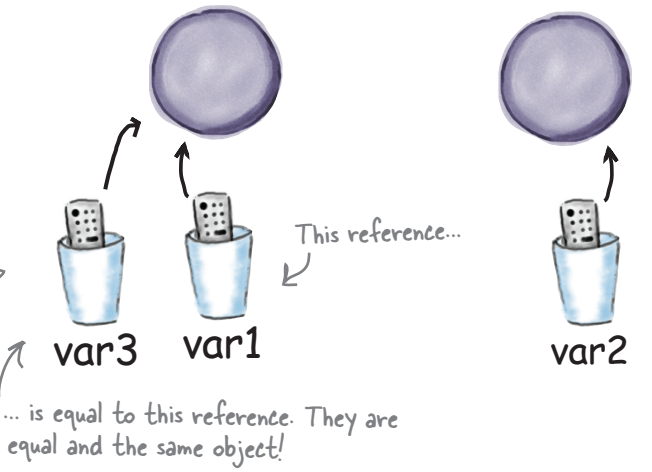


Two references are equal only if they reference the same object

The only way a test for equality between two variables containing object references returns true is when the two references point to the *same* object.

```
if (var1 === var3) {
    // wow, these are the same object!
}
```

Finally, two object references that are equal.



Sharpen your pencil

Here's a little code that helps find cars in Earl's Autos parking lot. Trace through this code and write the values of loc1 through loc4 below.

```
function findCarInLot(car) {
    for (var i = 0; i < lot.length; i++) {
        if (car === lot[i]) {
            return i;
        }
    }
    return -1;
}
var chevy = {
    make: "Chevy",
    model: "Bel Air"
};
var taxi = {
    make: "Webville Motors",
    model: "Taxi"
};
var fiat1 = {
    make: "Fiat",
    model: "500"
};
var fiat2 = {
    make: "Fiat",
    model: "500"
};

var lot = [chevy, taxi, fiat1, fiat2];

var loc1 = findCarInLot(fiat2);  _____
var loc2 = findCarInLot(taxi);   _____
var loc3 = findCarInLot(chevy);  _____
var loc4 = findCarInLot(fiat1);  _____
```

Your answers here.

Earl, from Earl's Autos.

The truthy is out there...

That's right, we said truthy not truth. We'll say falsey too. What on earth are we talking about? Well, some languages are rather precise about true and false. JavaScript, not so much. In fact, JavaScript is kind of loose about true and false. How is it loose? Well, there are values in JavaScript that aren't true or false, but that are nevertheless treated as true or false in a conditional. We call these values truthy and falsey precisely because they aren't technically true or false, but they behave like they are (again, inside a conditional).

Now here's the secret to understanding truthy and falsey: *concentrate on knowing what is falsey, and then everything else you can consider truthy*. Let's look at some examples of using these falsey values in a conditional:

Some people write it "falsy."

```
var testThis;
if (testThis) {
   // do something
}
```

Okay that's weird, we know this variable will be undefined in the conditional test. Does this work? Is this legal JavaScript? (Answer: yes.)

```
var element = document.getElementById("elementThatDoesntExist");
if (element) {
   // do something
}
```

Here the value of element is null. What's that going to do?

```
if (0) {
   // do another thing
}
```

We're testing 0?

Now we're doing a conditional test on an empty string. Anyone want to place bets?

```
if ("") {
   // does code here ever get evaluated? Place your bets.
}
```

Wait, now we're using NaN in a boolean condition? What's that going to evaluate to?

```
if (NaN) {
   // Hmm, what's NaN doing in a boolean test?
}
```

What JavaScript considers falsey

Again, the secret to learning what is truthy and what is falsey is to learn what's falsey, and then consider everything else truthy.

There are five falsey values in JavaScript:

undefined is falsey.

null is falsey.

0 is falsey.

The empty string is falsey.

NaN is falsey.

To remember which values are truthy and which are falsey, just memorize the five falsey values—undefined, null, 0, "" and NaN—and remember that everything else is truthy.

So, every conditional test on the previous page evaluated to false. Did we mention every other value is truthy (except for false, of course)? Here are some examples of truthy values:

This is an array. It's not undefined, null, zero, "" or NaN. It has to be true!

```
if ([]) {
   // this will happen
}

var element = document.getElementById("elementThatDoesExist");
if (element) {
   // so will this
}

if (1) {
   // gonna happen
}

var string = "mercy me";
if (string) {
   // this will happen too
}
```

This time we have an actual element object. That's not falsy either, so it's truthy.

Only the number 0 is falsey, all others are truthy.

Only the empty string is falsey, all other strings are truthy.

Sharpen your pencil

Time for a quick lie detector test. Figure out how many lies the perp tells, and whether the perp is guilty as charged, by determining which values are truthy and which values are falsey. Check your answer at the end of the chapter before you go on. And of course feel free to try these out in the browser yourself.

```
function lieDetectorTest() {
    var lies = 0;

    var stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    var car = {
        keysInPocket: null
    };
    if (car.keysInPocket) {
        console.log("Uh oh, guess you stole the car!");
        lies++;
    }
    if (car.emptyGasTank) {
        console.log("You drove the car after you stole it!");
        lies++;
    }
    var foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    var yourName = " ";  A string with one space.
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
var numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

What do you think this code does? Do you see anything odd about this code, especially given what we know about primitive types?

```
var text = "YOU SHOULD NEVER SHOUT WHEN TYPING";
var presentableText = text.toLowerCase();
if (presentableText.length > 0) {
     alert(presentableText);
}
```

The Secret Life of Strings

Types always belong to one of two camps: they're either a primitive type or an object. Primitives live out fairly simple lives, while objects keep state and have behavior (or said another way, have properties and methods). Right?

Well, actually, while all that is true, it's not the whole story. As it turns out, strings are a little more mysterious. Check out this code:

```
var emot = "XOxxOO";
var hugs = 0;
var kisses = 0;

emot = emot.trim();
emot = emot.toUpperCase();

for(var i = 0; i < emot.length ; i++) {
    if (emot.charAt(i) === "X") {
       hugs++;
    } else if (emot.charAt(i) == "O") {
       kisses++;
    }
}
```

This looks like a normal, primitive string.

Wait a sec, calling a method on a string?

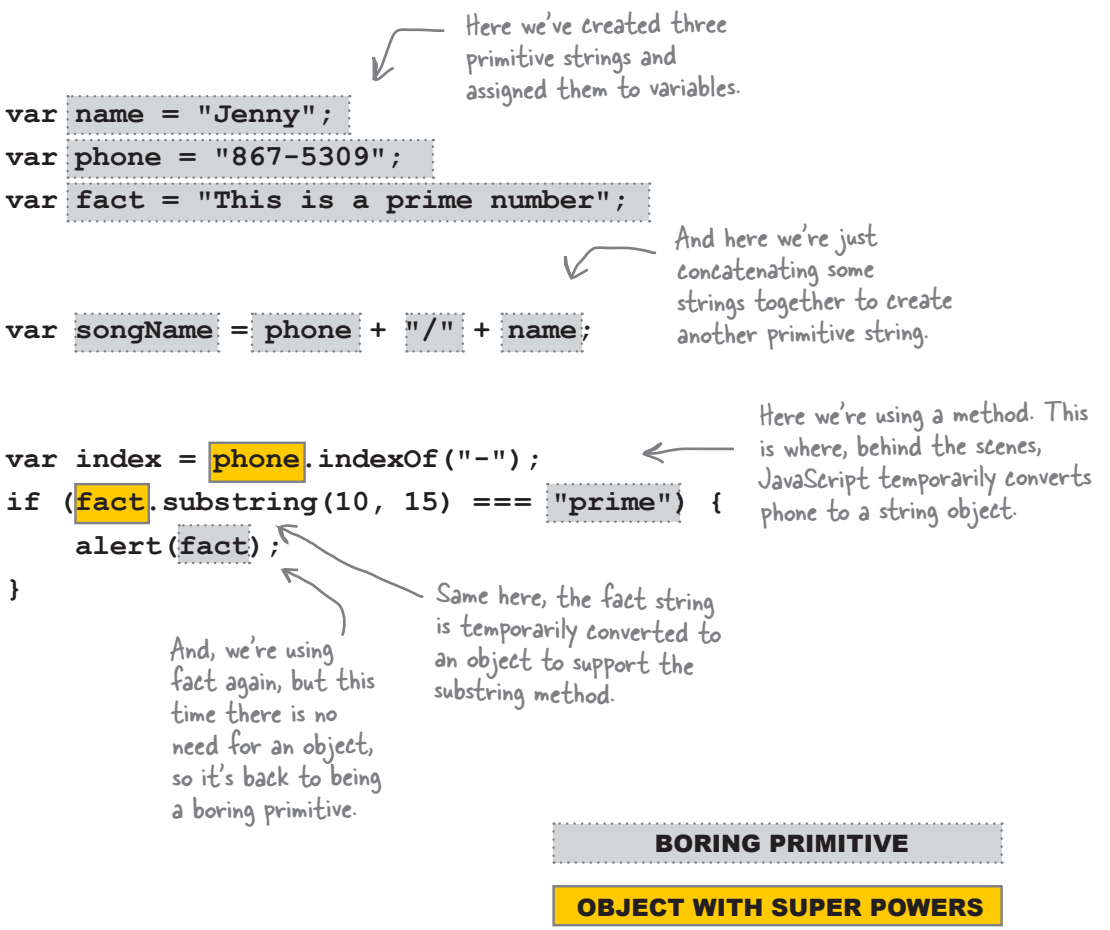
And a string with a property?

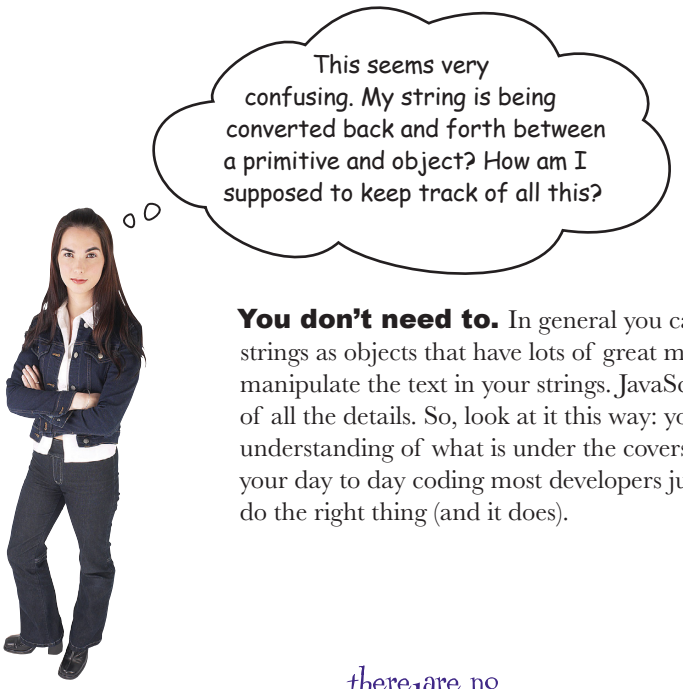
More methods?

How a string can look like a primitive and an object

How does a string masquerade as both a primitive and an object? Because JavaScript supports both. That is, with JavaScript you can create a string that is a primitive, and you can also create one that is an object (which supports lots of useful string manipulation methods). Now, we've never talked about how to create a string that is an object, and in most cases you don't need to explicitly do it yourself, because the JavaScript interpreter *will create string objects for you*, as needed.

Now, where and why might it do that? Let's look at the life of a string:





You don't need to. In general you can just think of your strings as objects that have lots of great methods to help you manipulate the text in your strings. JavaScript will take care of all the details. So, look at it this way: you now have a better understanding of what is under the covers of JavaScript, but in your day to day coding most developers just rely on JavaScript to do the right thing (and it does).

there are no Dumb Questions

Q: Just making sure, do I ever have to keep track of where my string is a primitive and where it's an object?

A: Most of the time, no. The JavaScript interpreter will handle all the conversion for you. You just write your code, assuming a string supports the object properties and methods, and things will work as expected.

Q: Why does JavaScript support a string as both a primitive and an object?

A: Think about it this way: you get the efficiency of the simple string primitive type as long as you are doing basic string operations like comparison, concatenation, writing string to the DOM, and so on. But if you need to do more sophisticated string processing, then you have the string object quickly at your disposal.

Q: Given an arbitary string, how do I know if it is an object or primitive?

A: A string is always a primitive unless you create it in a special way using an object constructor. We'll talk about object constructors later. And you can always use the typeof operator on your variable to see if it is of type string or object.

Q: Can other primitives act like objects?

A: Yes, numbers and booleans can also act like objects at times. However, neither of these has nearly as many useful properties as strings do, so you won't find you'll use this feature nearly as often as you do with strings. And remember, this all happens for you behind the scenes, so you don't really have to think about it much. Just use a property if you need to and let JavaScript handle the temporary conversion for you.

Q: How can I know all the methods and properties that are available for String objects?

A: That's where a good reference comes in handy. There are lots of online references that are helpful, and if you want a book, *JavaScript: The Definitive Guide* has a reference guide with information about every string property and method in JavaScript. Google works pretty well too.

A five-minute tour of string methods (and properties)

Given that we're in the middle of talking about strings and you've just discovered that strings also support methods, let's take a little break from talking about weirdo types and look at a few of the more common string methods you might want to use. A few string methods get used over and over, and it is highly worth your time to get to know them. So on with the tour.

A little pep talk: we could pull you aside and write an entire chapter on every method and property that strings support. Not only would that make this book 40 lbs and 2000 pages long, but at this point, you really don't need it—you already get the basics of methods and objects, and all you need is a good reference if you really want to dive into the details of string processing.

the length property

The length property holds the number of characters in the string. It's quite handy for iterating through the characters of the string.

We use the length property to iterate over each character in the string.

```
var input = "jenny@wickedlysmart.com";
for(var i = 0; i < input.length; i++) {
    if (input.charAt(i) === "@") {
        console.log("There's an @ sign at index " + i);
    }
}
```

And the charAt method to get the character at a particular index in the string.

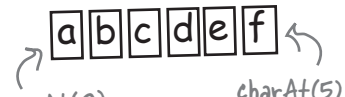
JavaScript console

```
There's an @ sign at index 5
```

the charAt method

The charAt method takes an integer number between zero and the length of the string (minus one), and returns a string containing the single character at that position of the string. Think of the string a bit like an array, with each character at an index of the string, with the indices starting at 0 (just like an array). If you give it an index that is greater than or equal to the length of the string, it returns the empty string.

Note that JavaScript doesn't have a character type. So characters are returned as new strings containing one character.



the indexOf method

This method takes a string as an argument and returns the index of the first character of the first occurrence of that argument in the string.

Here's the string we're going to call indexOf on.

```
var phrase = "the cat in the hat";
```

And our goal is to find the first occurence of "cat" in phrase.

```
var index = phrase.indexOf("cat");
console.log("there's a cat sitting at index " + index);
```

The index of the first cat is returned.

JavaScript console

```
There's a cat sitting at index 4
```

You can also add a second argument, which is the starting index for the search.

```
index = phrase.indexOf("the", 5);
console.log("there's a the sitting at index " + index);
```

Because we're starting the search at index 5, we're skipping the first "the" and finding the second "the" at index 11.

JavaScript console

```
There's a the sitting at index 11
```

```
index = phrase.indexOf("dog");
console.log("there's a dog sitting at index " + index);
```

Note if the string can't be found, then -1 is returned as the index.

JavaScript console

```
There's a dog sitting at index -1
```

the substring method

Give the substring method two indices, and it will extract and return the string contained within them.

```
var data = "name|phone|address";
var val = data.substring(5, 10);
console.log("Substring is " + val);
```

Here's the string we're going to call substring on.

We'd like the string from index 5 and up to (but not including) 10 returned.

We get back a new string with the characters from index 5 to 10.

```
JavaScript console
Substring is phone
```

You can omit the second index and substring will extract a string that starts at the first index and then continues until the end of the original string.

```
val = data.substring(5);
console.log("Substring is now " + val);
```

```
JavaScript console
Substring is now phone|address
```

the split method

The split method takes a character that acts as a delimiter, and breaks the string into parts based on the delimiter.

```
var data = "name|phone|address";
var vals = data.split("|");
console.log("Split array is ", vals);
```

Split uses the delimiter to break the original string into pieces, which are returned in an array.

Notice here we're passing two arguments to console.log separated by a comma. This way, the vals array doesn't get converted to a string before it's displayed in the console.

```
JavaScript console
Split array is ["name", "phone", "address"]
```

String Soup


toLowerCase
Returns a string with all uppercase characters changed to lowercase.
replace
Finds substrings and replaces them with another string.
lastIndexOf
Just like indexOf, but finds the last, not the first, occurrence.
Returns a new string that has part of the original string removed.
slice
Joins strings together.
concat
match
Searches for matches in a string using regular expressions.
Returns a portion of a string.
substring
Returns a string with all lowercase characters changed to uppercase.
trim
Removes whitespace from around the string. Handy when processing user input.
toUpperCase
There's really no end to learning all the things you can do with strings. Here are a few more methods available to you. Just get a passing familiarity right now, and when you really need them you can look up the details...

Chair Wars

(or How Really Knowing Types Can Change Your Life)

Once upon a time in a software shop, two programmers were given the same spec and told to "build it." The Really Annoying Project Manager forced the two coders to compete, by promising that whoever delivers first gets one of those cool Aeron™ chairs all the Silicon Valley guys have. Brad, the hardcore hacker scripter, and Larry, the college grad, both knew this would be a piece of cake.

Larry, sitting in his cube, thought to himself, "What are the things this code has to *do*? It needs to make sure the string is long enough, it needs to make sure the middle character is a dash, and it needs to make sure every other character is a number. I can use the string's `length` property and I know how to access its characters using the `charAt` method."

Brad, meanwhile, kicked back at the cafe and thought to himself, "What are the things this code has to do?" He first thought, "A string is an object, and there are lots of methods I can use to help validate the phone number. I'll brush up on those and get this implemented quickly. After all, an object is an object." Read on to see how Brad and Larry built their programs, and for the answer to your burning question: ***who got the Aeron?***

The spec

Take a phone number of the form:

"123-4567"

and write code to accept or reject it. To be accepted the number should have seven digits, 0 through 9, with a dash in the middle.

The chair

In Larry's cube

Larry set about writing code based on the string methods. He wrote the code in no time:

```
function validate(phoneNumber) {
    if (phoneNumber.length !== 8) {
       return false;
    }
    for (var i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.charAt(i) !== '-') {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

Larry uses the length property of the string object to see how many characters it has.

He uses the charAt method to examine each character of the string.

First, he makes sure character three has a dash.

Then he makes sure each character zero through two and four through six has a number in it.

In Brad's cube

Brad wrote code to check for two numbers and a dash:

```
function validate(phoneNumber) {
   if (phoneNumber.length !== 8) {
      return false;
   }
   var first = phoneNumber.substring(0,3);
   var second = phoneNumber.substring(4);
   if (phoneNumber.charAt(3) !== "-" || isNaN(first) || isNaN(second)) {
      return false;
   }
   return true;
}
```

Brad starts just like Larry...

But he uses his knowledge of the string methods.

He uses the substring method to create a string containing three characters from zero up to character three.

And again to start at character index four up to the end of the string.

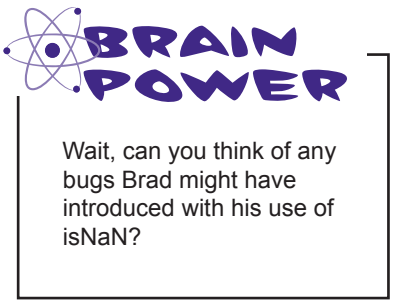
Then he tests all the conditions for being a correct phone number in one conditional.

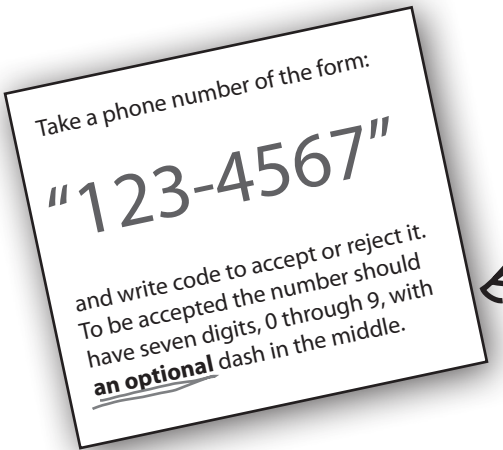
And interestingly, knowing it or not, he's depending on some type conversions here to convert a string to a number, and then making sure it's a number with isNaN. Clever!

But wait! There's been a spec change.

"Okay, *technically* you were first, Larry, because Brad was looking up how to use all those methods," said the Manager, "but we have to add just one tiny thing to the spec. It'll be no problem for crack programmers like you two."

"If I had a dime for every time I've heard *that* one", thought Larry, knowing that spec-change-no-problem was a fantasy. "And yet Brad looks strangely serene. What's up with that?" Still, Larry held tight to his core belief that Brad's fancy way, while cute, was just showing off. And that he'd win again in this next round and produce the code first.





What got added to the spec

Back in Larry's cube

Larry thought he could use most of his existing code; he just had to work these edge cases of the missing dash in the number. Either the number would be only seven digits, or it would be eight digits with a dash in the third position. Quickly Larry coded the additions (which took a little testing to get right):

```
function validate(phoneNumber) {
    if (phoneNumber.length  > 8 ||
          phoneNumber.length < 7) {
       return false;
    }
    for (var i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.length === 8 &&
                    phoneNumber.charAt(i) !== '-') {
              return false;
           } else if (phoneNumber.length === 7 &&
                    isNaN(phoneNumber.charAt(i))) {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

Larry had to make a few additions to his logic. Not a lot of code, but it's getting a bit hard to decipher.

At Brad's laptop at the beach

Brad smiled, sipped his margarita and quickly made his changes. He simply got the second part of the number using the length of the phone number minus four as the starting point for the substring, instead of hardcoding the starting point at a position that assumes a dash. That almost did it, but he did need to rewrite the test for the dash because it applies only when the phone number has a length of eight.

About the same number of changes as Larry, but Brad's code is still easier to read.

```
function validate(phoneNumber) {
    if (phoneNumber.length  > 8 ||
        phoneNumber.length < 7) {
        return false;
    }
    var first = phoneNumber.substring(0,3);
    var second = phoneNumber.substring(phoneNumber.length - 4);

    if (isNaN(first) || isNaN(second)) {
       return false;
    }
    if (phoneNumber.length === 8) {
       return (phoneNumber.charAt(3) === "-");
    }
    return true;
}
```

Now Brad's getting the second number using the total length of the phone number to get the starting point.

And he's validating the dash only if the number is eight characters.

Here we're returning the result of evaluating the conditional, which will be true or false.

Err, we think Brad still has a bug. Can you find it?

How would you rewrite Brad's code to use the split method instead?

Larry snuck in just ahead of Brad.

But the smirk on Larry's face melted when the Really Annoying Project Manager said, "Brad, your code is very readable and maintainable. Good job."

But Larry shouldn't be too worried, because, as we know, there is more than just code beauty at work. This code still needs to get through QA, and we're not quite sure Brad's code works in all cases. What about you? Who do you think deserves the chair?

The suspense is killing me. Who got the chair?

Amy from the second floor.

(Unbeknownst to all, the Project Manager had given the spec to *three* programmers.)

Wow, a one-liner! Check out how this works in the appendix!

Here's Amy's code.

```
function validate(phoneNumber) {
    return phoneNumber.match(/^\d{3}-?\d{4}$/);
}
```

IN THE LABORATORY, AGAIN

The lab crew continues to probe JavaScript using the **typeof** operator and they're uncovering some more interesting things deep within the language. In the process, they've discovered a new operator, **instanceof**. With this one, they're truly on the cutting edge. Put your lab coat and safety goggles back on and see if you can help decipher this JavaScript and the results. *Warning: this is definitely going to be the weirdest code you've seen so far.*

Here's the code. Read it, run it, alter it, massage it, see what it does...

How strange. Doesn't this look a bit like a mix of a function and an object?

```
function Duck(sound) {
    this.sound = sound;
    this.quack = function() {console.log(this.sound);}
}

var toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Hmm "new". We've haven't seen that before. But we're guessing we should read this as, create a new Duck and assign it to the toy variable.

If it looks like an object, and walks like an object... let's test it.

Okay, and here is instanceof...

Be sure to check your output with the answers at the end of the chapter. But just what does this all mean? Ah, we'll be getting to all that in just a couple of chapters. And, in case you didn't notice, you are well on your way to being a pretty darn advanced JavaScript coder. This is serious stuff!

JavaScript console

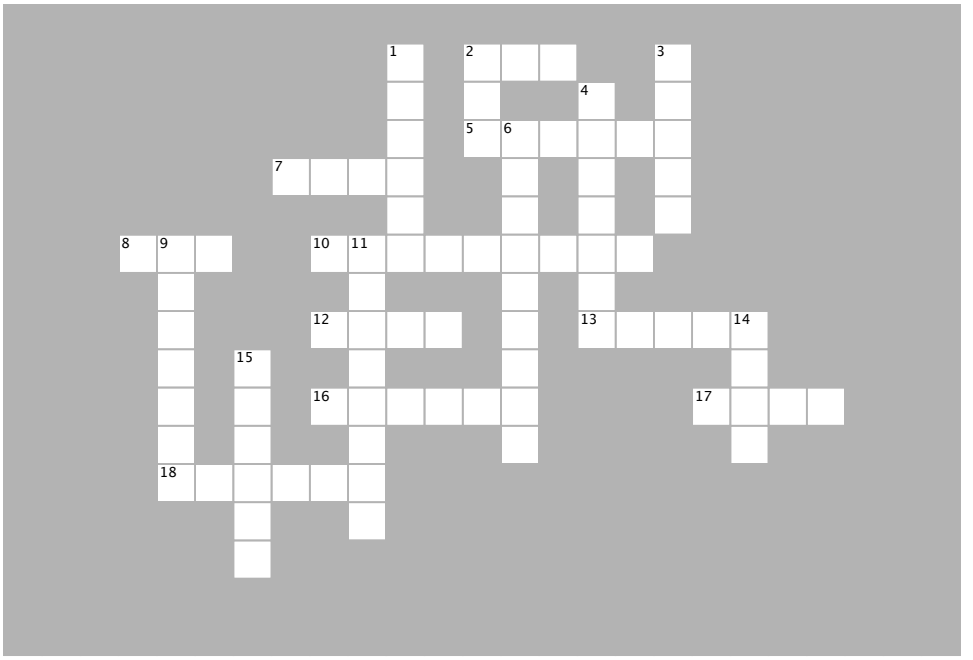
Put your results here. Are there any surprises?

BULLET POINTS

- There are two groups of types in JavaScript: **primitives** and objects. Any value that isn't a primitive type is an **object**.
- The primitives are: numbers, strings, booleans, null and undefined. Everything else is an object.
- **undefined** means that a variable (or property or array item) hasn't yet been initialized to a value.
- **null** means "no object".
- "NaN" stands for "Not a Number", although a better way to think of **NaN** is as a number that can't be represented in JavaScript. The type of NaN is number.
- NaN never equals any other value, including itself, so to test for NaN use the function **isNaN**.
- Test two values for equality using == or ===.
- If two operands have different types, the equality operator (==) will try to convert one of the operands into another type before testing for equality.
- If two operands have different types, the strict equality operator (===) returns false.
- You can use === if you want to be sure no type conversion happens, however, sometimes the type conversion of == can come in handy.
- Type conversion is also used with other operators, like the arithmetic operators and string concatenation.
- JavaScript has five **falsey** values: undefined, null, 0, "" (the empty string) and false. All other values are **truthy**.
- Strings sometimes behave like objects. If you use a property or method on a primitive string, JavaScript will convert the string to an object temporarily, use the property, and then convert it back to a primitive string. This happens behind the scenes so you don't have to think about it.
- The string has many methods that are useful for string manipulation.
- Two objects are equal only if the variables containing the object references point to the same object.

JavaScript cross

You're really expanding your JavaScript skills. Do a crossword to help it all sink in. All the answers are from this chaper.



ACROSS

2. The only value in JavaScript that doesn't equal anything.
5. The type of Infinity is _____.
7. There are _____ falsey values in JavaScript.
8. Who got the Aeron?
10. Two variables containing object references are equal only if they _____ the same object.
12. The value returned when you're expecting an object, and that object doesn't exist.
13. The _____ method is a string method that returns an array.
16. It's always 67 degrees in _____, Missouri.
17. The type of null in the JavaScript specification.
18. The _____ equality operator returns true only if the operands have the same type and the same value.

DOWN

1. The _____ operator can be used to get the type of a value.
2. The weirdest value in the world.
3. Your Fiat is parked at _____ Autos.
4. Sometimes strings masquerade as _____.
6. The value of a property that doesn't exist.
9. There are lots of handy string _____ you can use.
11. The _____ operator tests two values to see if they're equal, after trying to convert the operands to the same type.
14. null == undefined
15. To find a specific character at an index in a string, use the _____ method.

A bunch of JavaScript values and party crashers, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Fill in the blank next to each sentence with the name of one attendee. We've already guessed one of them.

Here's our solution:

Tonight's attendees:

I get returned from a function when there is no return statement. → undefined

I'm the value of a variable when I haven't been assigned a value. → undefined

I'm the value of an array item that doesn't exist in a sparse array. → undefined

I'm the value of a property that doesn't exist. → undefined

I'm the value of a property that's been deleted. → undefined

I'm the value that can't be assigned to a property when you create an object. → undefined

zero

empty object

null

undefined

NaN

infinity

area 51

...---...

{}

[]

IN THE LABORATORY SOLUTION

In the laboratory we like to take things apart, look under the hood, poke and prod, hook up our diagnostic tools and check out what is really going on. Today, we're investigating JavaScript's type system and we've found a little diagnostic tool called **typeof** to examine variables. Put your lab coat and safety goggles on, and come on in and join us.

The **typeof** operator is built into JavaScript. You can use it to probe the type of its operand (the thing you use it to operate on). Here's an example:

```
var subject = "Just a string";

var probe = typeof subject;
console.log(probe);
```

The typeof operator takes an operand, and evaluates to the type of the operand.

The type here is "string". Note that typeof uses strings to represent types, like "string", "boolean", "number", "object", "undefined" and so on.

JavaScript console

```
string
```

Now it's your turn. Collect the data for the following experiments:

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var test8 = ["abcdef", 123];
function test9(){return "abcdef"};

console.log(typeof test1);
console.log(typeof test2);
console.log(typeof test3);
console.log(typeof test4);
console.log(typeof test5);
console.log(typeof test6);
console.log(typeof test7);
console.log(typeof test8);
console.log(typeof test9);
```

Here's the test data, and the tests.

JavaScript console

```
string
number
boolean
object
object
undefined
object
object
function
```

Here are our results.

BACK IN THE LABORATORY SOLUTION

Oops, we forgot null in our test data. Here's the missing test case:

```
var test10 = null;

console.log(typeof test10);
```

Here's our result.

JavaScript console

```
object
```

Exercise Solution

We've been looking at some rather, um, interesting, values so far in this chapter. Now, let's take a look at some interesting behavior. Try adding the code below to the <script> element in a basic web page and see what you get in the console when you load up the page. You won't get why yet, but see if you can take a guess about what might be going on.

```
if (99 == "99") {
    console.log("A number equals a string!");
} else {
    console.log("No way a number equals a string");
}
```

Here's what we got.

JavaScript console

```
A number equals a string!
```

Sharpen your pencil Solution

For each comparison below write true or false below the operators == or === to represent the result of the comparison:

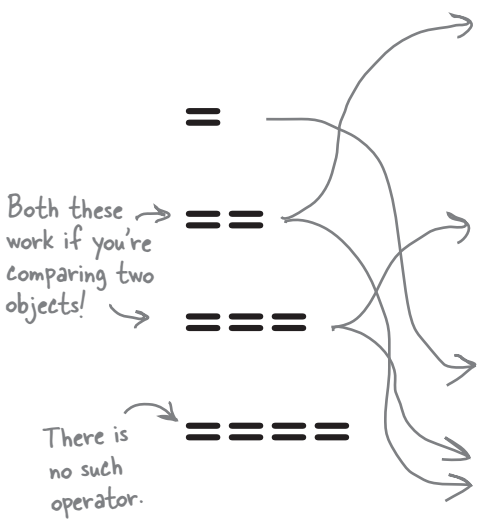
	==	===	
`"42" == 42`	true	false	`"42" === 42`
`"0" == 0`	true	false	`"0" === 0`
`"0" == false`	true	false	`"0" === false`
`"true" == true`	false	false	`"true" === true`
`true == (1 == "1")`	true	false	`true === (1 === "1")`

Tricky!

If you replace both == with ===, then the result is false.

WHO DOES WHAT? SOLUTION

We had our descriptions for these operators all figured out, and then they got all mixed up. Can you help us figure out who does what? Be careful, we're not sure if each contender matches zero, one or more descriptions. Here's our solution:



Compares values to see if they are equal. This is the considerate equality operator. He'll go to the trouble of trying to convert your types to see if you are really equal.

Compares values to see if they are equal. This guy won't even consider values that have different types.

Assigns a value to a variable.

Compares object references and returns true if they are the same and false otherwise.

Sharpen your pencil Solution

For each expression below, write the result in the blank next to it. We've done one for you. Here's our solution.

```
Infinity - "1"                    Infinity
```
"1" is converted to 1, and Infinity – 1 is Infinity.

```
"42" + 42                         "4242"
```

```
2 + "1 1"                         "21 1"
```

```
99 + 101                          200
```

```
"1" - "1"                         0
```
Both strings are converted to 1, and 1–1 is 0.

```
console.log("Result: " + 10/2)    "Result: 5"
```
10/2 happens first, and the result is concatenated to the string "Result: "

```
3 + " bananas " + 2 + " apples"   "3 bananas 2 apples"
```
Each + is concatenation because for both, one operand is a string.

Sharpen your pencil Solution

Time for a quick lie detector test. Figure out how many lies the perp tells, and whether the perp is guilty as charged, by determining which values are truthy and which values are falsey. Here's our solution. Did you try these out in the browser yourself?

```
function lieDetectorTest() {
    var lies = 0;

    var stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    var car = {
        keysInPocket: null
    };
    if (car.keysInPocket) {
        console.log("Uh oh, guess you stole the car!");
        lies++;
    }
    if (car.emptyGasTank) {
        console.log("You drove the car after you stole it!");
        lies++;
    }
    var foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    var yourName = " ";
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
var numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

Any object is truthy, even an empty one.

This perp didn't steal the car because the value of the keysInPocket property is null, which is falsey.

And the perp didn't drive the car either, because the emptyGasTank property is undefined, which is falsey.

But [ ] (an empty array) is truthy, so the perp was caught on the scene.

There is no item in the array, so the array item at 0 is undefined, which is falsey. Hmm, the perp must have hidden the stash already.

A string with one space.

Any non-empty string is truthy, even if it just has one space!

The number of lies is 3 so we think the perp is guilty.

JavaScript console

```
You stole the diamond
A sure sign of guilt
Guess you lied about your name
You told 3 lies!
Guilty as charged
```

Sharpen your pencil Solution

Here's a little code that helps find cars in Earl's Used Autos parking lot. Trace through this code and write the values of loc1 through loc4 below.

```
function findCarInLot(car) {
    for (var i = 0; i < lot.length; i++) {
        if (car === lot[i]) {
            return i;
        }
    }
    return -1;
}
var chevy = {
    make: "Chevy",
    model: "Bel Air"
};
var taxi = {
    make: "Webville Motors",
    model: "Taxi"
};
var fiat1 = {
    make: "Fiat",
    model: "500"
};
var fiat2 = {
    make: "Fiat",
    model: "500"
};

var lot = [chevy, taxi, fiat1, fiat2];

var loc1 = findCarInLot(fiat2);
var loc2 = findCarInLot(taxi);
var loc3 = findCarInLot(chevy);
var loc4 = findCarInLot(fiat1);
```

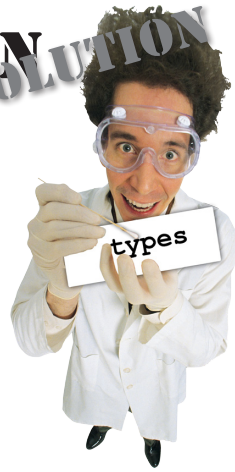
Here are our answers.

3
1
0
2

Earl, from Earl's Autos.

IN THE LABORATORY, AGAIN

SOLUTION



The lab crew continues to probe JavaScript using the **typeof** operator and they're uncovering some more interesting things deep within the language. In the process, they've discovered a new operator, **instanceof**. With this one, they're truly on the cutting edge. Put your lab coat and safety goggles back on and see if you can help decipher this JavaScript and the results. *Warning: this is definitely going to be the weirdest code you've seen so far.*

Here's the code. Read it, run it, alter it, massage it, see what it does...

How strange. Doesn't this look a bit like a mix of a function and an object?

```
function Duck(sound) {
    this.sound = sound;
    this.quack = function() {console.log(this.sound);}
}

var toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Hmm "new". We've haven't seen that before. But we're guessing we should read this as, create a new Duck and assign it to the toy variable.

If it looks like an object, and walks like an object... let's test it.

Okay, and here is instanceof...

Just what does this all mean? Ah, we'll be getting to all that in just a few chapters. And, in case you didn't notice, you are well on your way to being a pretty darn advanced JavaScript coder. This is serious stuff!

JavaScript console

quack quack ← The toy acts like an object... we can call its method.

object ← And the type is object.

true ← But it is an "instanceof" a Duck, whatever that means... Hmm.

↑ Here are our results.

JavaScript cross Solution

You're really expanding your JavaScript skills. Do a crossword to help it all sink in. All the answers are from this chaper. Here's our solution.