

4 putting some order in your data

Arrays

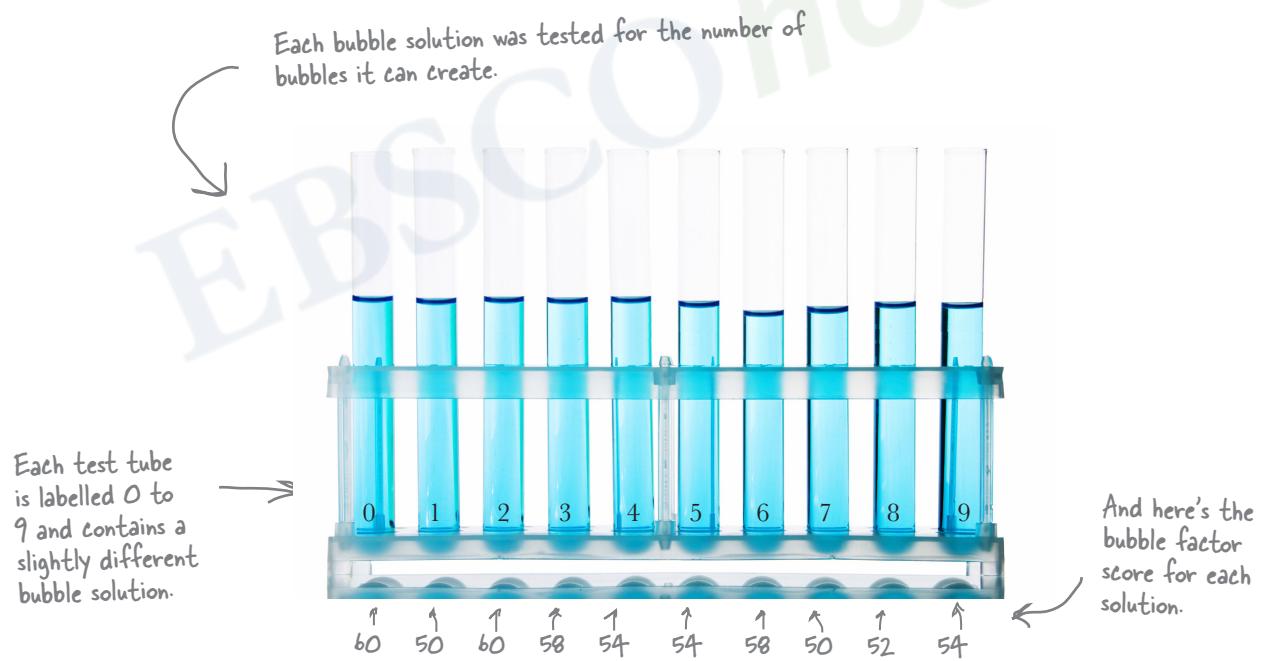


There's more to JavaScript than numbers, strings and booleans. So far you've been writing JavaScript code with **primitives**—simple strings, numbers and booleans, like "Fido", 23, and true. And you can do a lot with primitive types, but at some point you've got to deal with **more data**. Say, all the 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-U's?

Meet the Bubbles-R-U's 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:



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, 54, 58, 50, 52, 54];
```

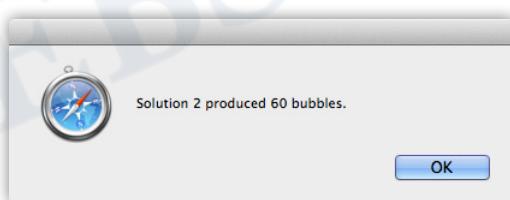
↑
Here's all ten values, grouped together into an array, and assigned to the `scores` variable.

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.

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

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]`.



My
bubble solution #2
is definitely going to be
the best.



One of the
Bubbles-R-U's
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"];
```

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

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

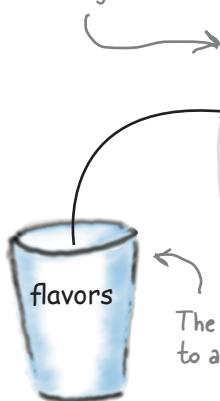
and then list each item of the array...

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

... 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:

The array collects all these values together.



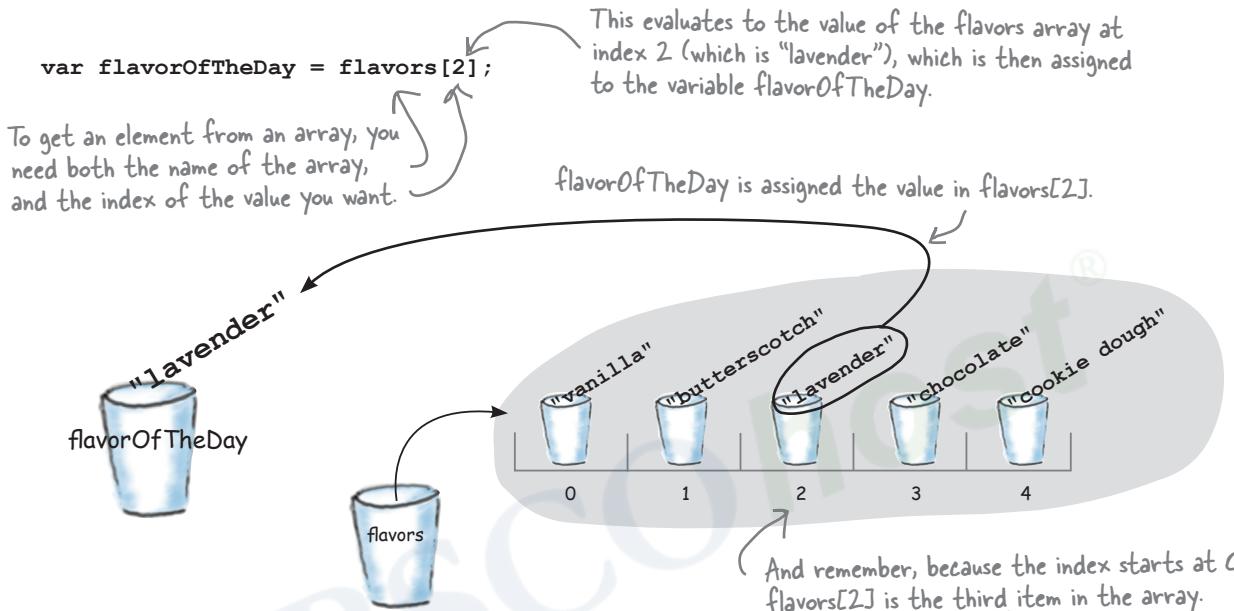
The array is assigned to a variable.

Each value has an index number, starting at 0.

Every index in the array holds a value.

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:



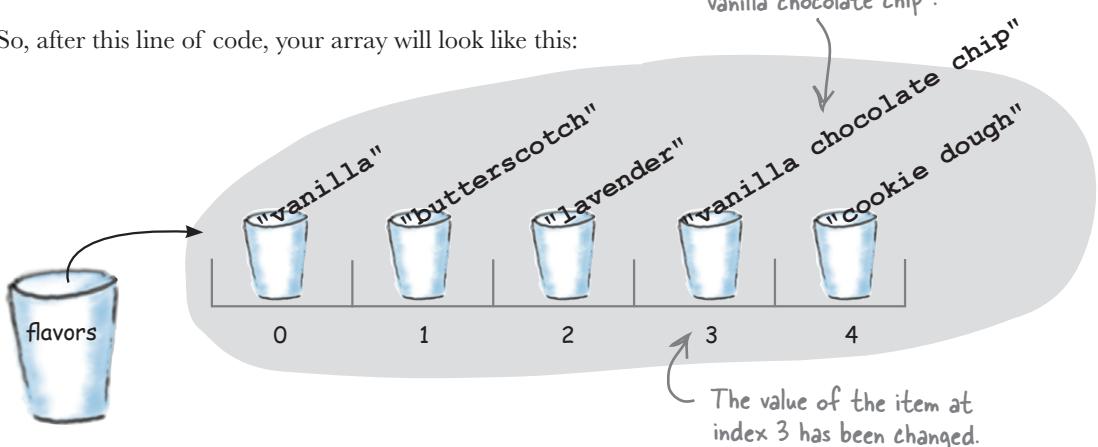
Updating a value in the array

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

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

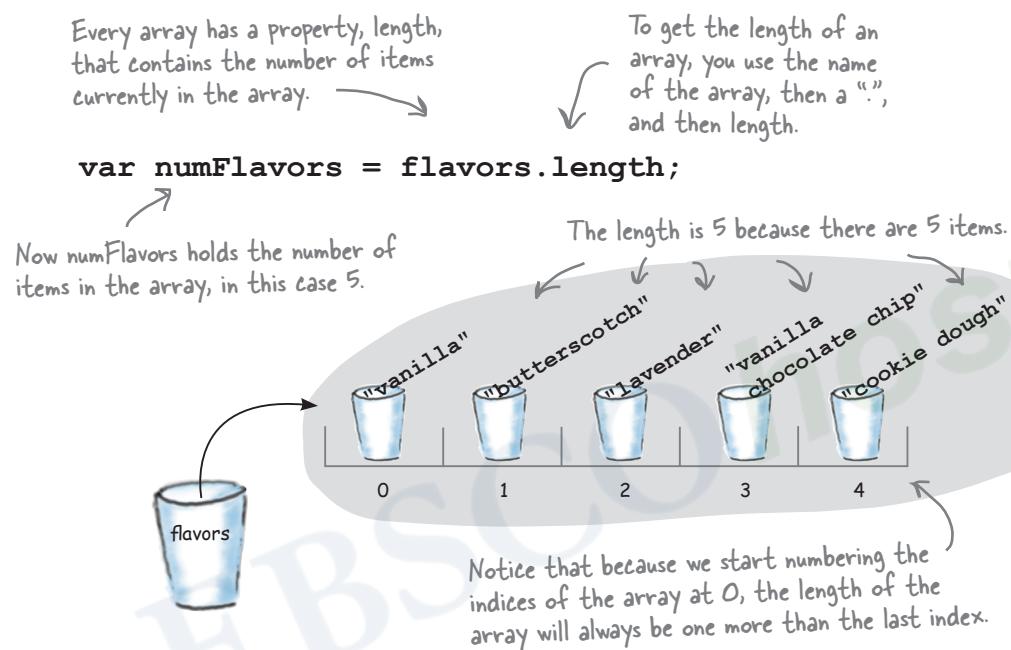
This sets the value of the item at index 3 (previously "chocolate") to a new value, "vanilla chocolate chip".

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



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:



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];
```



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>
```

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.

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:

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

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

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...

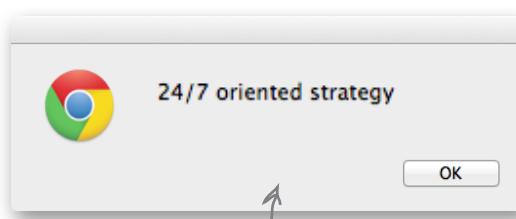
```
var phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
```

- 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.

```
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];
```

The Bubbles-R-Us CEO

What we need to build.

New bubble scores.

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

thinking about the bubble scores report

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

We need to start by listing all the solutions and their corresponding scores.

Then we need to print the total number of bubble scores.

Followed by the highest score and each solution that has that score.

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

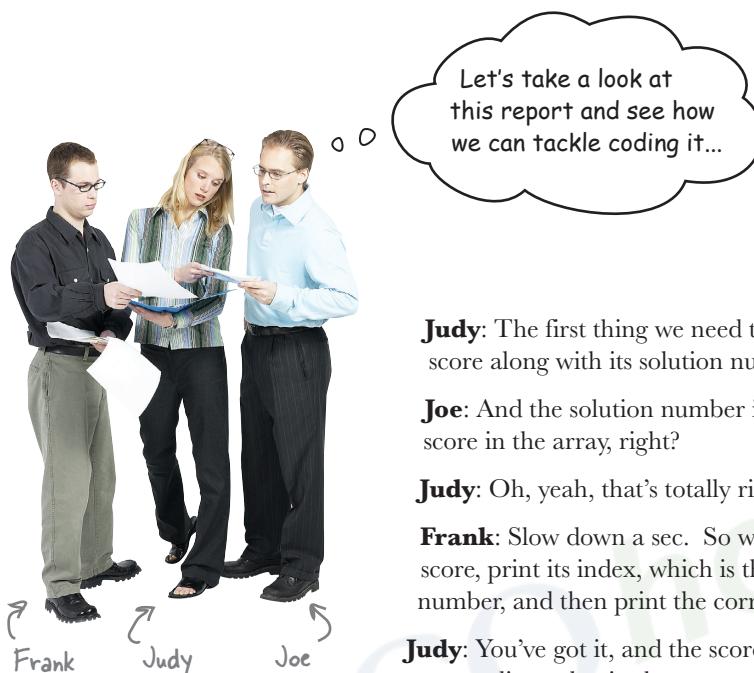
Highest bubble score: 69

Solutions with highest score: #11, #18



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.

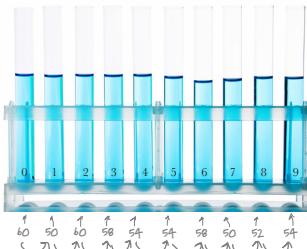
Judy: That works for me! Let's do it, and then we'll come back to the rest of the report.

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

var i = 0; Create a variable to keep track of the current index.

while (i < scores.length) { And keep looping while our index is less than the length of the array.

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

```
    console.log(output);
```

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

```
}
```

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

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.

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.

Rearrange the magnets here.

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

```
{ } i = i + 2;
} } i = i + 1;
} { { }
```

```
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");
```

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.

```

A var i = 0;           ↗ First we INITIALIZED a counter.
while B i < scores.length { ↗ Then we tested that counter in a CONDITIONAL expression.
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    C i = i + 1;          ↗ We also had a BODY to execute; that is, all the
}                         statements between the { and }.
                        ↗ And finally, we INCREMENTED the counter.

```

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

```

A for loop starts with the keyword for. ↗ In the parentheses, there are three parts. The first part is the loop variable INITIALIZATION. This initialization happens only once, before the for loop starts.
for A (var i = 0; i < scores.length; C i = i + 1) { ↗ The second part is the CONDITIONAL test. Each time we loop, we perform this test, and if it is false, we stop.
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
}                         ↗ And the third part is where we INCREMENT the counter. This happens once per loop, after all the statements in the BODY.

```

The BODY goes here. Notice there are no changes other than moving the increment of i into the for statement.



```
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;
}
```

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.

Your code goes here. ↴

We've got all the pieces for the first part of the report, let's put this all together...



```

<!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>

```

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

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.

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.



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
```



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.

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

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



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:

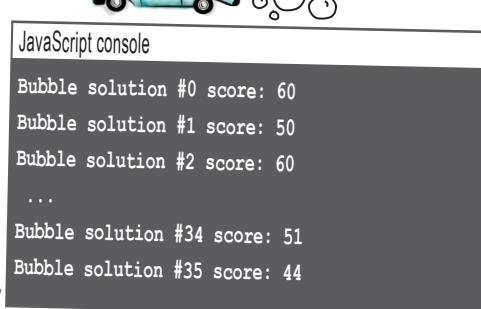
```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];
for (var i = 0; i < scores.length; i++) {
    var output = "Bubble solution #" + 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 iterate through the array. Here, let me write some pseudocode:

```

DECLARE a variable highScore and set to zero. ← Add a variable to hold the high score.

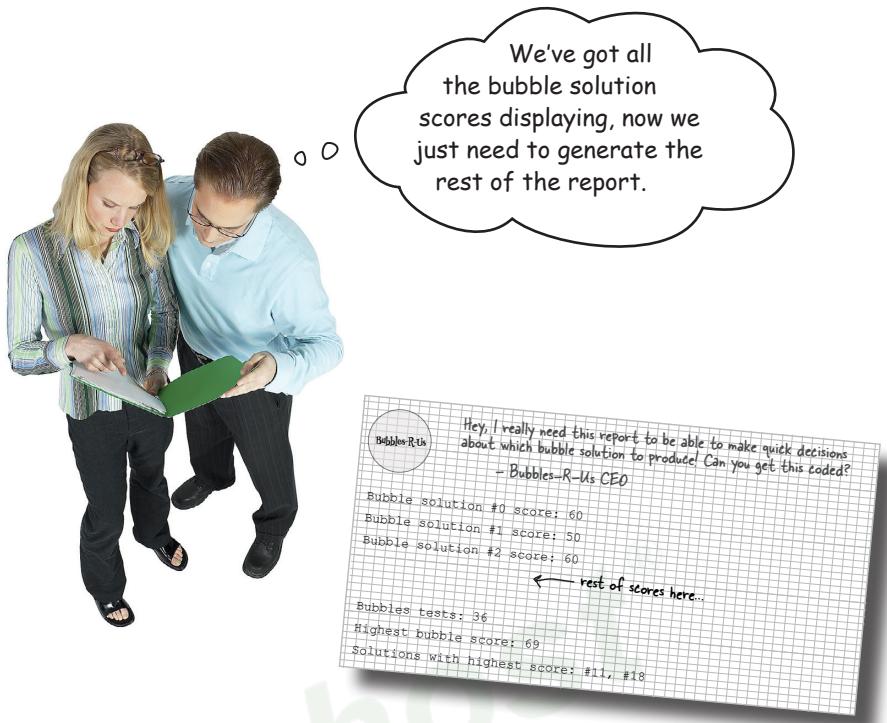
FOR: var i=0; i < scores.length; i++
    DISPLAY the bubble solution score[i]
    IF scores[i] > highScore
        SET highScore = scores[i];
    END IF
END FOR
DISPLAY highScore ← After the loop we just display the high score.

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

```

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: _____
```



Hey you guys are almost there! All you've got left is collecting up all the solutions with highest score and printing them. Remember, there might be more than one.

"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.

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;
sparseArray[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!

Now that we know how to add items to an array we can finish up this report. We can just create the array of the solutions with the highest score as we iterate through the scores array to find the highest bubble score, right?



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.

```
var bestSolutions = [];
```

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

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

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.

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

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



We spent a
lot of time in the last
chapter talking about functions.
How come we're not using any?

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:

```

<!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>

```

Here's the Bubbles-R-U's code.



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.



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) { ← Okay, here's the new
            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>
```

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*:

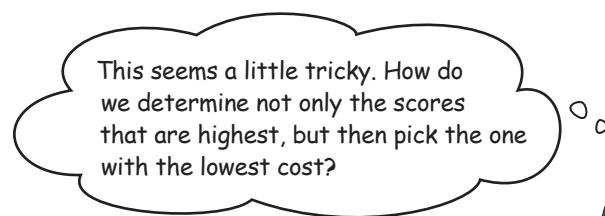
Scores and costs are parallel arrays because for each score there is a corresponding cost at the same index.

```
var costs = [.25, .27, .25, .25, .25, .25, .33, .31, .25, .29, .27, .22, ..., .29];
```

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.

```
var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69, ..., 44];
```



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) {
```

```
}
```

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

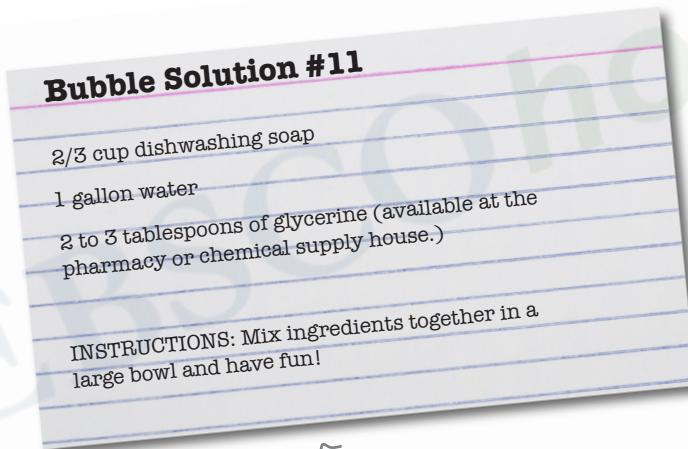
← Translate the
pseudocode to
JavaScript here.

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!



DO try this at HOME!





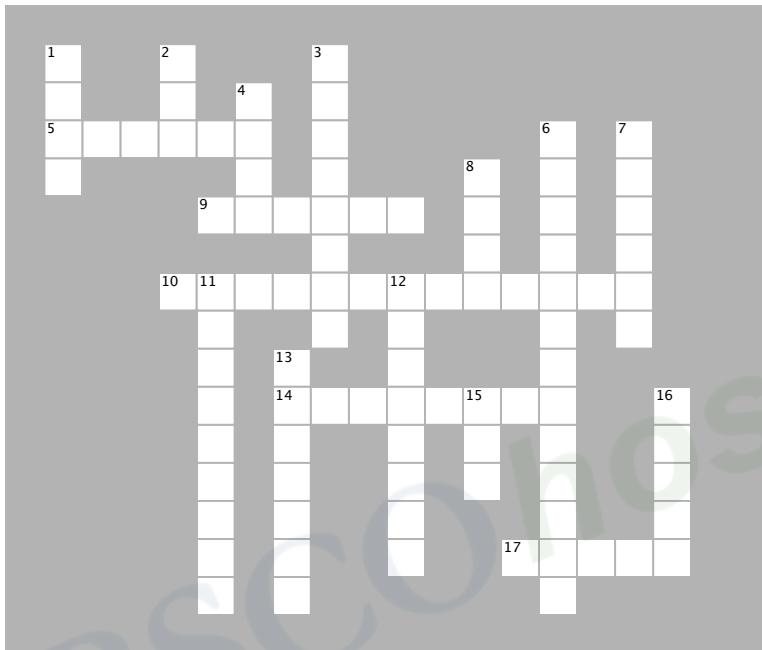
BULLET POINTS

- 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;
```

```
}
```

Leftover magnets.

```
{     i = i + 2;
```

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

Here's the output
we're expecting.

JavaScript console

Bubblegum contains bubble gum!

Rearrange the magnets here.





Sharpen your pencil 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");
    }
}
```

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.



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.

```
function getMostCostEffectiveSolution(scores, costs, highscore) {
  var cost = 100;   ↪ We'll keep track of the lowest cost solution in cost...
  var index;       ↪ ... and the index of the lowest cost solution in index.

  for (var i = 0; i < scores.length; i++) {           ↪ We iterate through the scores
    if (scores[i] == highscore) {                      ↪ ... and check to see if the score has the high score.

      if (cost > costs[i]) {                         ↪ If it does, then we can check its cost. If the current cost is
        index = i;                                     greater than the solution's cost, then we've found a lower cost
        cost = costs[i];                                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
    }
  }
  return index;
}

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

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

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.

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

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

↪ 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.



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

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
```



JavaScript cross Solution

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

