# Arrays

The array is the most common data structure in computer programming. Every language has some form of array built into the language. Because arrays are built-in, they are usually very efficient and can be considered good choices for certain data collection purposes. In this chapter we explore how arrays work in JavaScript and when to use them.

## JavaScript Arrays Defined

The standard definition for an array is a linear collection of elements, where the elements can be accessed via indices, which are generally integers used to compute offsets. Most computer programming languages have these types of arrays. JavaScript, on the other hand, has a different type of array.

A JavaScript array is actually a specialized type of JavaScript object, where the indices are property names that can be integers used to represent offsets. However, when integers are used for indices, they are converted to strings internally in order to conform to the requirements for JavaScript objects. Because JavaScript arrays are really just objects, they are not quite as efficient at the arrays of other programming languages.

While JavaScript arrays are, strictly speaking, JavaScript objects, they are specialized objects that are actually categorized internally as arrays. The Array is one of the recognized JavaScript object types, and as such, there is a set of properties and methods you can use with arrays.

## Using Arrays

Arrays in JavaScript are very flexible. There are several different ways to create arrays, access array elements, and perform tasks such as searching and sorting using arrays. The newest version of JavaScript also includes array methods that allow programmers to work with arrays using functional programming techniques. We cover all these different topics in this section.

### Creating Arrays

The simplest way to create an array is by declaring an array variable using the [] operator:

var numbers = [];

When you create an array in this fashion, you have an array with a length of 0. You can verify this fact by calling one of the built-in properties of the Array object, the length property:

console.log(numbers.length); // displays 0

Another way to create an array is to declare an array variable with a set of elements inside the [] operator:

var numbers = [1,2,3,4,5];

console.log(numbers.length); // displays 5

You can also create an array by calling the Array constructor:

var numbers = new Array()

console.log(numbers.length); // displays 0

You can create an array using the Array constructor and specify the number of elements for the array:

You can call the Array constructor with a set of elements as arguments to the constructor:

var numbers = new Array(1,2,3,4,5)

console.log(numbers.length); // displays 5

Finally, you can create an array by calling the Array constructor with a single argument specifying the length of the array:

var numbers = new Array(10);

console.log(numbers.length); // displays 10

Unlike many other programming languages, but common for most scripting languages, JavaScript array elements do not have to all be of the same type:

var objects = [1,"Joe",true,null];

We can verify that an object is an array by calling the Array.isArray() method, like this:

var numbers = [1,2,3,4,5];

if (Array.isArray(numbers)) {

// perform array tasks on numbers

}

As for which array creation technique is best, most JavaScript experts say using the [] operator is more efficient than calling the Array constructor (Flanagan, Chapter 7, p. 142; Crockford, Chapter 6, p. 58).

### Accessing and Writing Array Elements

Data are assigned to array elements using the [] operator in an assignment statement. For example, the following loop assigns the values 1-100 to an array:

var nums = []

for (var i = 0; i < 100; ++i) {

nums[i] = i+1;

}

Array elements are accessed using the [] operator. For example:

var numbers = [1,2,3,4,5];

var sum = numbers[0] + numbers[1] + numbers[2] + numbers[3] +

numbers[4];

console.log(sum); // displays 15

Of course, accessing all the array elements sequentially is much easier using a for loop:

var numbers = [1,2,3,4,5];

var sum = 0;

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

sum += numbers[i];

}

console.log(sum); // displays 15

Notice that the for loop is controlled using the length property rather than an integer literal. Because JavaScript arrays are objects, they can grow beyond the size specified when they were created, so it is always safest to use the length property when accessing all the elements of an array.

### Creating Arrays From Strings

Arrays can be created as the result of calling the split() method on a string. This method breaks up a string at a common delimiter, such as a space for each word, and creates an array consisting of the individual parts of the string.

The following short program demonstrates how the split() method works on a simple string:

var sentence = "the quick brown fox jumped over the lazy dog";

var words = sentence.split(" ");

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

print("word " + i + ": " + words[i]);

}

The output from this program is:

word 0: the

word 1: quick

word 2: brown

word 3: fox

word 4: jumped

word 5: over

word 6: the

word 7: lazy

word 8: dog

### Aggregate Array Operations

There are several operations you can perform on arrays aggregately. First, you can assign the data stored in one array to another array:

var nums = []

for (var i = 90; i <= 100; ++i) {

nums[i] = i;

}

var samenums = nums;

However, when you assign one array to another array, you are assigning a reference to the assigned array. When you make a change to the original array, that change is reflected in the other array as well. The following code fragment demonstrates how this works:

var nums = []

for (var i = 0; i < 100; ++i) {

nums[i] = i+1;

}

var samenums = nums;

nums[0] = 400;

print(samenums[0]); // displays 400

This is called a *shallow copy*. The new array's elements simply point to the original array's elements. A better alternative is to make a *deep copy*, so that each of the original array's elements is actually copied to the new array's elements. An effective way to do this is to create a function to perform this task:

function copy(arr1, arr2) {

for (var i = 0; i < arr1.length; ++i) {

arr2[i] = arr1[i];

}

}

Now the following code fragment produces the expected result:

var nums = []

for (var i = 0; i < 100; ++i) {

nums[i] = i+1;

}

var samenums = [];

copy(nums, samenums);

nums[0] = 400;

print(samenums[0]); // displays 1

Another aggregate operation you can perform is displaying the contents of an array using a function such as print(). For example:

var nums = [1,2,3,4,5];

print(nums);

will produce the following output:

1,2,3,4,5

This output may not be particularly useful, but you can use it to display the contents of an array when all you need is just a simple list.

## Accessor Methods

JavaScript provides a set of methods that you can use to access the elements of an array. These methods, called *accessor* methods, return some representation of the target array as their return values.

### Searching for a Value

One of the most commonly used accessor methods is indexOf(), which looks to see if the argument passed to the method is found in the array. If the argument is contained in the array, the method returns the index position of the argument. If the argument is not found in the array, the method returns -1. Here is an example:

var names =

["David","Cynthia","Raymond","Clayton","Mike","Jennifer","Bryan"];

putstr("Enter a name to search for: ");

var name = readline();

var position = names.indexOf(name);

if (position >= 0) {

print("Found " + name + " at position " + position);

}

else {

print(name + " not found in array.");

}

If you run this program and enter "Cynthia", the program will output:

Found Cynthia at position 1

If you enter "Joe", the output is:

Joe not found in array.

If you have multiple occurrences of the same data in an array, the indexOf() method will always return the position of the first occurrence. A similar method, lastIndexOf(), will, as it name suggests, return the position of the last occurrence of the argument in the array, or -1 if the argument isn't found. Here is an example:

var names = ["David","Mike","Cynthia","Raymond","Clayton","Mike","Jennifer"];

var name = "Mike";

var firstPos = names.indexOf(name);

print("First found " + name + " at position " + firstPos);

var lastPos = names.lastIndexOf(name);

print("Last found " + name + " at position " + lastPos);

The output from this program is:

First found Mike at position 1

Last found Mike at position 5

### String Representations of Arrays

There are two methods that return string representations of an array – join() and toString(). Both methods return a string that contains the elements of the array delimited by commas. Here are some examples:

var names = ["David","Cynthia","Raymond","Clayton","Mike","Jennifer"];

var namestr = names.join();

print(namestr); // David,Cynthia,Raymond,Clayton,Mike,Jennifer

namestr = names.toString();

print(namestr); // David,Cynthia,Raymond,Clayton,Mike,Jennifer

When you call the print() method with an array name, it automatically calls the toString() method for that array:

print(names); // David,Cynthia,Raymond,Clayton,Mike,Jennifer

### Creating New Arrays from Existing Arrays

There are two accessor methods that allow you to create new arrays from existing arrays: concat() and splice(). The concat() method allows you to put together two or more arrays to create a new array and the splice() method allows you to create a new array from a subset of an existing array.

Let's look first at how concat() works. The method is called from an existing array and its argument is another existing array. The argument array is concatenated to the end of the array calling concat(). The following program demonstrates how concat() works:

var cis = ["Mike","Clayton","Terrill","Danny","Jennifer"];

var dmp = ["Raymond","Cynthia","Bryan"];

var it = cis.concat(dmp);

print(it);

it = dmp.concat(cis);

print(it);

This program outputs:

Mike,Clayton,Terrill,Danny,Jennifer,Raymond,Cynthia,Bryan

Raymond,Cynthia,Bryan,Mike,Clayton,Terrill,Danny,Jennifer

The first output line shows the data from the data from the cis array first and the second output line shows the data from the dmp array first.

The splice() method creates a new array from the contents of an existing array. The arguments to the method are the starting position for taking the splice and the number of elements to take from the existing array. Here is how the method works:

var it = ["Mike","Clayton","Terrill","Raymond","Cynthia","Bryan",

"Danny","Jennifer"];

var dmp = it.splice(3,3);

var cis = it;

print(dmp); // Raymond,Cynthia,Bryan

print(cis); // Mike,Clayton,Terrill,Danny,Jennifer

## Mutator Methods

JavaScript has a set of *mutator* methods that allow you to modify the contents of an array without referencing the individual elements. These methods often make hard techniques easy, as you'll see below.

### Adding Elements to an Array

There are two mutator methods for adding elements to an array, push() and unshift(). The push() method adds an element to the end of an array:

var nums = [1,2,3,4,5];

print(nums); // 1,2,3,4,5

nums.push(6);

print(nums); // 1,2,3,4,5,6

Using push() is more intuitive than accomplishing the same result using the length property:

var nums = [1,2,3,4,5];

print(nums); // 1,2,3,4,5

nums[nums.length] = 6;

print(nums); // 1,2,3,4,5,6

Adding data to the beginning of an array is much harder than adding data to the end of an array. To do so without the benefit of a mutator method, each existing element of the array has to be shifted up one position before the new data is added. Here is some code to illustrate this scenario:

var nums = [2,3,4,5];

var newnum = 1;

var N = nums.length;

for (var i = N; i >= 0; --i) {

nums[i] = nums[i-1];

}

nums[0] = newnum;

print(nums); // 1,2,3,4,5

This code becomes more inefficient as the number of elements stored in the array increases.

The mutator method for adding array elements to the beginning of an array is unshift(). Here is how it works:

var nums = [2,3,4,5];

print(nums); // 2,3,4,5

var newnum = 1;

nums.unshift(newnum);

print(nums); // 1,2,3,4,5

nums = [3,4,5];

nums.unshift(newnum, 2);

print(nums); // 1,2,3,4,5

The second call to unshift() demonstrates that you can add multiple elements to an array with one call to the method.

### Removing Elements from an Array

Removing an element from the end of an array is easy using the pop() mutator method:

var nums = [1,2,3,4,5,9];

nums.pop();

print(nums); // 1,2,3,4,5

Without mutator methods, removing elements from the beginning of an array requires shifting elements towards the beginning of the array, causing the same inefficiency we see when adding elements to the beginning of an array:

var nums = [9,1,2,3,4,5];

print(nums);

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

nums[i] = nums[i+1];

}

print(nums); // 1,2,3,4,5,

Besides the fact that we have to shift the elements down to collapse the array, we are also left with an extra element. We know this because of the extra comma we see when we display the array contents.

The mutator method we need to remove an element from the beginning of an array is shift(). Here is how it works:

var nums = [9,1,2,3,4,5];

nums.shift();

print(nums); // 1,2,3,4,5

You'll notice there are no extra elements left at the end of the array.

### Adding and Removing Elements from the Middle of an Array

Trying to add or remove elements at the end of an array leads to the same problems we find when trying to add or remove elements from the beginning of an array – both operations require shifting array elements either toward the beginning or toward the end of the array. There is one mutator method, however, we can use to perform both operations – splice().

To add elements to an array using splice(), you have to provide the following arguments:

* The starting index (where you want to begin adding elements)
* The number of elements to remove (0 when you are adding elements)
* The elements you want to add to the array

Let's look at a simple example. The following program adds elements to the middle of an array:

var nums = [1,2,3,7,8,9];

var newElements = [4,5,6];

nums.splice(3,0,newElements);

print(nums); // 1,2,3,4,5,6,7,8,9

Here is an example of using splice() to remove elements from an array:

var nums = [1,2,3,100,200,300,400,4,5];

nums.splice(3,4);

print(nums); // 1,2,3,4,5

### Putting Array Elements in Order

The last two mutator methods are used to arrange array elements into some type of order. The first of these, reverse(), reverses the order of the elements of an array. Here is an example of its use:

var nums = [1,2,3,4,5];

nums.reverse();

print(nums); // 5,4,3,2,1

A task we want to perform quite often on the elements in an array is sorting the elements into order. The mutator method for this task, sort(), works very well with strings:

var names = ["David","Mike","Cynthia","Clayton","Bryan","Raymond"];

nums.sort();

print(nums); // Bryan,Clayton,Cynthia,David,Mike,Raymond

But sort() does not work so well with numbers:

var nums = [3,1,2,100,4,200];

nums.sort();

print(nums); // 1,100,2,200,3,4

The sort() method sorts its data lexicographically, assuming the data elements are strings, even though in this case the elements are numbers. We can make the sort() method work correctly for numbers by passing in an ordering function to the method, which the method will then use to sort the array elements. For numbers, the ordering function can simply subtract one number from another number. If the number returned is negative, the left operand is less than the right operand; if the number returned is zero, the left operand is equal to the right operand; if the number returned is positive, the left operand is greater than the right operand.

With this in mind, let's rerun the small program above using an ordering function:

function compare(num1, num2) {

return num1 - num2;

}

var nums = [3,1,2,100,4,200];

nums.sort(compare);

print(nums); // 1,2,3,4,100,200

The sort() method uses the compare() function to sort the array elements numerically rather than lexicographically.

## Iterator Methods

The final set of methods we are going to examine are called *iterator* methods. These methods apply a function to each element of an array, either returning a value, or a set of values, or a new array after applying the function to each element.

### Non-Array-Generating Methods

The first group of methods we are going to discuss do not generate a new array, but either simply perform an operation on each element of an array, or generate a single value from an array.

The first of these methods is forEach(). This method takes a function as an argument and applies the called function to each element of an array. Here is an example of how the method works:

function square(num) {

print(num, num \* num);

}

var nums = [];

for (var i = 0; i < 10; ++i) {

nums[i] = i+1;

}

nums.forEach(square);

The output from this program is:

1 1

2 4

3 9

4 16

5 25

6 36

7 49

8 64

9 81

10 100

The next method, every(), applies a Boolean function (predicate) to an array and returns true if the function can return true for every element in the array. Here is how the every() method works:

function isEven(num) {

return num % 2 == 0;

}

var nums = [2,4,6,8,10];

var even = nums.every(isEven);

if (even) {

print("all numbers are even");

}

else {

print("some numbers are odd");

}

This program displays:

all numbers are even

If we change the array to:

var nums = [2,4,6,7,8,10];

the program displays:

some numbers are odd

The some() method will take a boolean function and return true if at least one of the elements in the array meets the criterion of the boolean function. Here is an example:

function isEven(num) {

return num % 2 == 0;

}

var nums = [1,2,3,4,5,6,7,8,9,10];

var someEven = nums.some(isEven);

if (someEven) {

print("some numbers are even");

}

else {

print("no numbers are even");

}

nums = [1,3,5,7,9];

var someEven = nums.some(isEven);

if (someEven) {

print("some numbers are even");

}

else {

print("no numbers are even");

}

The output from this program is:

some numbers are even

no numbers are even

The reduce() method applies a function to two successive elements of an array until the end of the array is reached, yielding a single value. Here is an example of using reduce() to compute the sum of the elements of an array:

function add(num1, num2) {

return num1 + num2;

}

var nums = [1,2,3,4,5,6,7,8,9,10];

var sum = nums.reduce(add);

print(sum); // displays 55

The reduce() method, in conjunction with the add() function, works from left-to-right, computing a running sum of the elements in the array, like this:

add(1,2) -> 3

add(3,3) -> 6

add(6,4) -> 10

add(10,5) -> 15

add(15,6) -> 21

add(21,7) -> 28

add(28,8) -> 36

add(36,9) -> 45

add(45,10) -> 55

We can also use reduce() with strings to perform “functional” concatenation:

function concat(str1, str2) {

return str1 + str2;

}

var words = ["the ", "quick ","brown ", "fox "];

var sentence = words.reduce(concat);

print(sentence); // displays "the quick brown fox"

JavaScript also provides a reduceRight() method which works similarly to reduce(), only working from the right-hand side of the array to the left, rather than from left-to-right. The following program is used to reverse the elements of an array using reduceRight():

function concat(str1, str2) {

return str1 + str2;

}

var words = ["the ", "quick ","brown ", "fox "];

var sentence = words.reduceRight(concat);

print(sentence); // displays "fox brown quick the"

### Iterator Methods That Return a New Array

There are two iterator methods that return new arrays – map() and filter(). The map() method works like the forEach() method, applying a function to each element of an array. The difference between the two methods is map() returns a new array with the results of the function application. Here is an example:

function curve(grade) {

return grade += 5;

}

var grades = [77, 65, 81, 92, 83];

var newgrades = grades.map(curve);

print(newgrades); // 82, 70, 86, 97, 88

Here is an example using strings:

function first(word) {

return word[0];

}

var words = ["for","your","information"];

var acronym = words.map(first);

print(acronym.join("")); // displays "fyi"

For this last example, the acronym array stores the first letter of each word in words. However, if we want to display the elements of the array as a true acronym, we need to get rid of the commas that will be displayed if we just display the array elements using the implied toString() method. We accomplish this by calling the join() method with the empty string as the separator.

The filter() method works similarly to every(), but instead of returning true if all the elements of an array satisfy a Boolean function, the method returns a new array consisting of those elements that satisfy the Boolean function. Here is an example:

function isEven(num) {

return num % 2 == 0;

}

function isOdd(num) {

return num % 2 != 0;

}

var nums = [];

for (var i = 0; i < 20; ++i) {

nums[i] = i+1;

}

var evens = nums.filter(isEven);

print("Even numbers: ");

print(evens);

var odds = nums.filter(isOdd);

print("Odd numbers: ");

print(odds);

This program returns the following output:

Even numbers:

2,4,6,8,10,12,14,16,18,20

Odd numbers:

1,3,5,7,9,11,13,15,17,19

Here's another interesting use of filter():

function passing(num) {

return num >= 60;

}

var grades = [];

for (var i = 0; i < 20; ++i) {

grades[i] = Math.floor(Math.random() \* 101);

}

var passGrades = grades.filter(passing);

print(grades);

print(passGrades);

This program displays:

39,43,89,19,46,54,48,5,13,31,27,95,62,64,35,75,79,88,73,74

89,95,62,64,75,79,88,73,74

Of course, we can also use filter() with strings. Here is an example that applies the spelling rule "i before e except after c":

function afterc(str) {

if (str.indexOf("cie") > -1) {

return true;

}

return false;

}

var words = ["recieve","deceive","percieve","deceit","concieve"];

var misspelled = words.filter(afterc);

print(misspelled); // displays receive,perceive,conceive

## Two-Dimensional and Multi-Dimensional Arrays

JavaScript arrays are only one-dimensional, but you can create multi-dimensional arrays by creating arrays of arrays. In this section we'll describe how to create two-dimensional arrays in JavaScript.

### Creating Two-Dimensional Arrays

A two-dimensional array is structured like a spreadsheet with rows and columns. To create a two-dimensional array in JavaScript, we have to create an array and then make each element of that array an array also. At the very least, we need to know the number of rows we want the array to contain. With that information, we can create a two-dimensional array with *n* rows and one column:

var twod = [];

var rows = 5;

for (var i = 0; i < rows; ++i) {

twod[i] = [];

}

The problem with this approach is that each element of the array is set to undefined. A better way to create a two-dimensional array is to follow the example from (Crockford, page 64). Crockford extends the JavaScript Array object with a function that sets the number of rows and columns and sets each value to a value passed to the function. Here is his definition:

Array.matrix = function(numrows, numcols, initial) {

var arr = [];

for (var i = 0; i < numrows; ++i) {

var columns = [];

for (var j = 0; j < numcols; ++j) {

columns[j] = initial;

}

arr[i] = columns;

}

return arr;

}

Here is some code to test the definintion:

var nums = Array.matrix(5,5,0);

print(nums[1][1]); // displays 0

var names = Array.matrix(3,3,"");

names[1][2] = "Joe";

print(names[1][2]); // display "Joe"

Of course, we can also create a two-dimensional array and initialize it to a set of values in one line:

var grades = [[89, 77, 78],[76, 82, 81],[91, 94, 89]];

print(grades[2][2]); // displays 89

For small data sets, this is the easiest way to create a two-dimensional array.

### Processing Two-Dimensional Array Elements

There are two fundamental patterns used to process the elements of a two-dimensional array. One pattern emphasizes accessing array elements by columns and the other pattern emphasizes accessing array elements by rows. We will use the grades array created in the last code segment above to demonstrate how these patterns work.

For both patterns we use a set of nested for loops. For columnar processing, the outer loop moves through the rows and the inner loop processes the columns. For the grades array, think of each row as a set of grades for one student. We can compute the average of that student's grades by adding each grade on the row to a total variable and then dividing by the total number of grades. Here is the code for that process:

var grades = [[89, 77, 78],[76, 82, 81],[91, 94, 89]];

var total = 0;

var average = 0.0;

for (var row = 0; row < grades.length; ++row) {

for (var col = 0; col < grades[row].length; ++col) {

total += grades[row][col];

}

average = total / grades[row].length;

print("Student " + parseInt(row+1) + " average: " +

average.toFixed(2));

total = 0;

average = 0.0;

}

The inner loop is controlled by the expression:

col < grades[row].length

This works because each row contains an array and that array has a length property we can use to determine how many columns there are in the row.

The grade average for each student is rounded to two decimals using the toFixed(n) method.

Here is the output from the program:

Student 1 average: 81.33

Student 2 average: 79.67

Student 3 average: 91.33

To perform a row-wise computation, we simply have to flip the for loops, so that the outer loop controls the columns and the inner loop controls the rows. Here is the new program:

var grades = [[89, 77, 78],[76, 82, 81],[91, 94, 89]];

var total = 0;

var average = 0.0;

for (var col = 0; col < grades.length; ++col) {

for (var row = 0; row < grades[col].length; ++row) {

total += grades[row][col];

}

average = total / grades[col].length;

print("Test " + parseInt(col+1) + " average: " +

average.toFixed(2));

total = 0;

average = 0.0;

}

The output from this program is:

Test 1 average: 85.33

Test 2 average: 84.33

Test 3 average: 82.67

### Jagged Arrays

A jagged array is an array where not all the columns in the array are equal. One row may have 3 columns, while another row may have 5 columns, while yet another row only has 1 column. Many programming languages have problems handling jagged arrays, but JavaScript does not since we can compute the length of any row.

To give you an example, imagine the grades array where students have an unequal number of grades recorded. We can still compute the correct average for each student without changing the program at all:

var grades = [[89, 77],[76, 82, 81],[91, 94, 89, 99]];

var total = 0;

var average = 0.0;

for (var row = 0; row < grades.length; ++row) {

for (var col = 0; col < grades[row].length; ++col) {

total += grades[row][col];

}

average = total / grades[row].length;

print("Student " + parseInt(row+1) + " average: " +

average.toFixed(2));

total = 0;

average = 0.0;

}

Notice that the first student only has 2 grades, while the second student has 3 grades, and the third student has 4 grades. Because the program computes the length of the row in the inner loop, this jaggedness doesn't cause any problems. Here is the output from this program:

Student 1 average: 83.00

Student 2 average: 79.67

Student 3 average: 93.25

## Arrays of Objects

All of the examples in this chapter have consisted of arrays whose elements have been primitive data types, such as numbers and strings. Arrays can also contain objects and all the methods and properties of arrays work with objects.

function Point(x,y) {

this.x = x;

this.y = y;

}

function displayPts(arr) {

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

print(arr[i].x + ", " + arr[i].y);

}

}

var p1 = new Point(1,2);

var p2 = new Point(3,5);

var p3 = new Point(2,8);

var p4 = new Point(4,4);

var points = [p1,p2,p3,p4];

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

print("Point " + parseInt(i+1) + ": " + points[i].x + ", " +

points[i].y);

}

var p5 = new Point(12,-3);

points.push(p5);

displayPts(points);

points.shift();

displayPts(points);

The output from this program is:

Point 1: 1, 2

Point 2: 3, 5

Point 3: 2, 8

Point 4: 4, 4

1, 2

3, 5

2, 8

4, 4

12, -3

3, 5

2, 8

4, 4

12, -3

The point 12, -3 is added to the array using push() and the point 1,2 is removed from the array using shift().

## Arrays in Objects

We can use arrays to store complex data in an object. Many of the data structures we study in this book are implemented as class objects with an underlying array to store data.

The following example demonstrates many of the techniques we use throughout the book. In the example, we create an object that stores the weekly observed high temperature. The object has methods for adding a new temperature and for computing the average of the temperatures stored in the object. Here is the code:

function weekTemps() {

this.dataStore = [];

this.add = add;

this.average = average;

}

function add(temp) {

this.dataStore.push(temp);

}

function average() {

var total = 0;

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

total += this.dataStore[i];

}

return total / this.dataStore.length;

}

var thisWeek = new weekTemps();

thisWeek.add(52);

thisWeek.add(55);

thisWeek.add(61);

thisWeek.add(65);

thisWeek.add(55);

thisWeek.add(50);

thisWeek.add(52);

thisWeek.add(49);

print(thisWeek.average()); // displays 54.875

You'll notice the add() method uses the push() method to add elements to the dataStore array, using the name add() rather than push(). Using a more intuitive name for an operation is a common technique when defining object methods. Not everyone will understand what it means to push a data element, but everyone knows what it means to add a data element.

## Exercises

1. Create a grades object that stores a set of student grades in an object. Provide a method for adding a grade and a method for displaying the student's grade average.
2. Store a set of words in an array and display the contents both forwards and backwards.
3. Modify the weeklyTemps object so that it stores a month's worth of data using a two-dimensional array. Create methods to display the monthly average, a specific week's average, and all the week's averages.
4. Create an object that stores individual letters in an array and has a method for displaying the letters as a single word.