O'REILLY®

Data Structures & Algorithms with JavaScript

BRINGING CLASSIC COMPUTING APPROACHES TO THE WEB

Michael McMillan

O'REILLY®

Data Structures & Algorithms with JavaScript

As an experienced JavaScript developer moving to server-side programming, you need to implement classic data structures and algorithms associated with conventional object-oriented languages like C# and Java. This practical guide shows you how to work hands-on with a variety of storage mechanisms—including linked lists, stacks, queues, and graphs—within the constraints of the JavaScript environment.

Determine which data structures and algorithms are most appropriate for the problems you're trying to solve, and understand the tradeoffs when using them in a JavaScript program. An overview of the JavaScript features used throughout the book is also included.

This book covers:

- Arrays and lists: the most common data structures
- Stacks and queues: more complex list-like data structures
- Linked lists: how they overcome the shortcomings of arrays
- Dictionaries: storing data as key-value pairs
- Hashing: good for quick insertion and retrieval
- Sets: useful for storing unique elements that appear only once
- Binary Trees: storing data in a hierarchical manner
- Graphs and graph algorithms: ideal for modeling networks
- Algorithms: including those that help you sort or search data
- Advanced algorithms: dynamic programming and greedy algorithms

Michael McMillan is an instructor of Computer Information Systems at Pulaski Technical College in Arkansas, and an adjunct instructor of Information Science at the University of Arkansas at Little Rock. He was a programmer/analyst for Arkansas Children's Hospital.

JAVASCRIPT

US $34.99 CAN $36.99

ISBN: 978-1-449-36493-9

Twitter: @oreillymedia
facebook.com/oreilly

Data Structures and Algorithms with JavaScript

by Michael McMillan

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian MacDonald and Meghan Blanchette
Production Editor: Melanie Yarbrough
Copyeditor: Becca Freed
Proofreader: Amanda Kersey
Indexer: Ellen Troutman-Zaig

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Rebecca Demarest and Cynthia Clarke Fehrenbach

March 2014: First Edition

Revision History for the First Edition:

2014-03-06: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449364939* for release details.

ISBN: 978-1-449-36493-9

[LSI]

CHAPTER 1

The JavaScript Programming Environment and Model

This chapter describes the JavaScript programming environment and the programming constructs we'll use in this book to define the various data structures and algorithms examined.

The JavaScript Environment

JavaScript has historically been a programming language that ran only inside a web browser. However, in the past few years, there has been the development of JavaScript programming environments that can be run from the desktop, or similarly, from a server. In this book we use one such environment: the JavaScript shell that is part of Mozilla's comprehensive JavaScript environment known as SpiderMonkey.

To download the JavaScript shell, navigate to the Nightly Build web page. Scroll to the bottom of the page and pick the download that matches your computer system.

Once you've downloaded the program, you have two choices for using the shell. You can use it either in interactive mode or to interpret JavaScript programs stored in a file. To use the shell in interactive mode, type the command `js` at a command prompt. The shell prompt, `js>`, will appear and you are ready to start entering JavaScript expressions and statements.

The following is a typical interaction with the shell:

```
js> 1
1
js> 1+2
3
js> var num = 1;
js> num*124
124
```

```
js> for (var i = 1; i < 6; ++i) {
    print(i);
}
1
2
3
4
5
js>
```

You can enter arithmetic expressions and the shell will immediately evaluate them. You can write any legal JavaScript statement and the shell will immediately evaluate it as well. The interactive shell is great for exploring JavaScript statements to discover how they work. To leave the shell when you are finished, type the command `quit()`.

The other way to use the shell is to have it interpret complete JavaScript programs. This is how we will use the shell throughout the rest of the book.

To use the shell to intepret programs, you first have to create a file that contains a JavaScript program. You can use any text editor, making sure you save the file as plain text. The only requirement is that the file must have a *.js* extension. The shell has to see this extension to know the file is a JavaScript program.

Once you have your file saved, you interpret it by typing the `js` command followed by the full filename of your program. For example, if you saved the `for` loop code fragment that's shown earlier in a file named *loop.js*, you would enter the following:

```
c:\js>js loop.js
```

which would produce the following output:

```
1
2
3
4
5
```

After the program is executed, control is returned to the command prompt.

JavaScript Programming Practices

In this section we discuss how we use JavaScript. We realize that programmers have different styles and practices when it comes to writing programs, and we want to describe ours here at the beginning of the book so that you'll understand the more complex code we present in the rest of the book. This isn't a tutorial on using JavaScript but is just a guide to how we use the fundamental constructs of the language.

Declaring and Intializing Variables

JavaScript variables are global by default and, strictly speaking, don't have to be declared before using. When a JavaScript variable is initialized without first being declared, it becomes a global variable. In this book, however, we follow the convention used with compiled languages such as C++ and Java by declaring all variables before their first use. The added benefit to doing this is that declared variables are created as local variables. We will talk more about variable scope later in this chapter.

To declare a variable in JavaScript, use the keyword `var` followed by a variable name, and optionally, an assignment expression. Here are some examples:

```
var number;
var name;
var rate = 1.2;
var greeting = "Hello, world!";
var flag = false;
```

Arithmetic and Math Library Functions in JavaScript

JavaScript utilizes the standard arithmetic operators:

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- % (modulo)

JavaScript also has a math library you can use for advanced functions such as square root, absolute value, and the trigonometric functions. The arithmetic operators follow the standard order of operations, and parentheses can be used to modify that order.

Example 1-1 shows some examples of performing arithmetic in JavaScript, as well as examples of using several of the mathematical functions.

Example 1-1. Arithmetic and math functions in JavaScript

```
var x = 3;
var y = 1.1;
print(x + y);
print(x * y);
print((x+y)*(x-y));
var z = 9;
print(Math.sqrt(z));
print(Math.abs(y/x));
```

The output from this program is:

```
4.1
3.3000000000000003
7.789999999999999
3
0.3666666666666667
```

If you don't want or need the precision shown above, you can format a number to a fixed precision:

```
var x = 3;
var y = 1.1;
var z = x * y;
print(z.toFixed(2)); // displays 3.30
```

Decision Constructs

Decision constructs allow our programs to make decisions on what programming statements to execute based on a Boolean expression. The two decision constructs we use in this book are the `if` statement and the `switch` statement.

The `if` statement comes in three forms:

- The simple `if` statement
- The `if-else` statement
- The `if-else if` statement

Example 1-2 shows how to write a simple `if` statement.

Example 1-2. The simple `if` statement

```
var mid = 25;
var high = 50;
var low = 1;
var current = 13;
var found = -1;
if (current < mid) {
   mid = (current-low) / 2;
}
```

Example 1-3 demonstrates the `if-else` statement.

Example 1-3. The `if-else` statement

```
var mid = 25;
var high = 50;
var low = 1;
var current = 13;
var found = -1;
if (current < mid) {
   mid = (current-low) / 2;
}
```

```
else {
   mid = (current+high) / 2;
}
```

Example 1-4 illustrates the `if-else if` statement.

Example 1-4. The `if-else if` statement

```
var mid = 25;
var high = 50;
var low = 1;
var current = 13;
var found = -1;
if (current < mid) {
   mid = (current-low) / 2;
}
else if (current > mid) {
   mid = (current+high) / 2;
}
else {
   found = current;
}
```

The other decision structure we use in this book is the `switch` statement. This statement provides a cleaner, more structured construction when you have several simple decisions to make. Example 1-5 demonstrates how the `switch` statement works.

Example 1-5. The `switch` statement

```
putstr("Enter a month number: ");
var monthNum = readline();
var monthName;
switch (monthNum) {
   case "1":
      monthName = "January";
      break;
   case "2":
      monthName = "February";
      break;
   case "3":
      monthName = "March";
      break;
   case "4":
      monthName = "April";
      break;
   case "5":
      monthName = "May";
      break;
   case "6":
      monthName = "June";
      break;
   case "7":
```

```
        monthName = "July";
        break;
    case "8":
        monthName = "August";
        break;
    case "9":
        monthName = "September";
        break;
    case "10":
        monthName = "October";
        break;
    case "11":
        monthName = "November";
        break;
    case "12":
        monthName = "December";
        break;
    default:
        print("Bad input");
}
```

Is this the most efficient way to solve this problem? No, but it does a great job of demonstrating how the `switch` statement works.

One major difference between the JavaScript `switch` statement and `switch` statements in other programming languages is that the expression that is being tested in the statement can be of any data type, as opposed to an integral data type, as required by languages such as C++ and Java. In fact, you'll notice in the previous example that we use the month numbers as strings, rather than converting them to numbers, since we can compare strings using the `switch` statement in JavaScript.

Repetition Constructs

Many of the algorithms we study in this book are repetitive in nature. We use two repetition constructs in this book—the `while` loop and the `for` loop.

When we want to execute a set of statements while a condition is true, we use a `while` loop. Example 1-6 demonstrates how the `while` loop works.

Example 1-6. The `while` loop

```
var number = 1;
var sum = 0;
while (number < 11) {
    sum += number;
    ++number;
}
print(sum); // displays 55
```

When we want to execute a set of statements a specified number of times, we use a `for` loop. Example 1-7 uses a `for` loop to sum the integers 1 through 10.

Example 1-7. Summing integers using a `for` loop

```
var number = 1;
var sum = 0;
for (var number = 1; number < 11; number++) {
   sum += number;
}
print(sum); // displays 55
```

`for` loops are also used frequently to access the elements of an array, as shown in Example 1-8.

Example 1-8. Using a `for` loop with an array

```
var numbers = [3, 7, 12, 22, 100];
var sum = 0;
for (var i = 0; i < numbers.length; ++i) {
   sum += numbers[i];
}
print(sum); // displays 144
```

Functions

JavaScript provides the means to define both value-returning functions and functions that don't return values (sometimes called *subprocedures* or *void functions*).

Example 1-9 demonstrates how value-returning functions are defined and called in JavaScript.

Example 1-9. A value-returning function

```
function factorial(number) {
   var product = 1;
   for (var i = number; i >= 1; --i) {
      product *= i;
   }
   return product;
}

print(factorial(4)); // displays 24
print(factorial(5)); // displays 120
print(factorial(10)); // displays 3628800
```

Example 1-10 illustrates how to write a function that is used not for its return value, but for the operations it performs.

Example 1-10. A subprocedure or void function in JavaScript

```
function curve(arr, amount) {
   for (var i = 0; i < arr.length; ++i) {
      arr[i] += amount;
   }
}

var grades = [77, 73, 74, 81, 90];
curve(grades, 5);
print(grades); // displays 82,78,79,86,95
```

All function parameters in JavaScript are passed by value, and there are no reference parameters. However, there are reference objects, such as arrays, which are passed to functions by reference, as was demonstrated in Example 1-10.

Variable Scope

The *scope* of a variable refers to where in a program a variable's value can be accessed. The scope of a variable in JavaScript is defined as *function scope*. This means that a variable's value is visible within the function definition where the variable is declared and defined and within any functions that are nested within that function.

When a variable is defined outside of a function, in the main program, the variable is said to have *global* scope, which means its value can be accessed by any part of a program, including functions. The following short program demonstrates how global scope works:

```
function showScope() {
   return scope;
}

var scope = "global";
print(scope); // displays "global"
print(showScope()); // displays "global"
```

The function `showScope()` can access the variable `scope` because `scope` is a global variable. Global variables can be declared at any place in a program, either before or after function definitions.

Now watch what happens when we define a second `scope` variable within the `showScope()` function:

```
function showScope() {
   var scope = "local";
   return scope;
}

var scope = "global";
print(scope); // displays "global"
print(showScope()); // displays "local"
```

The `scope` variable defined in the `showScope()` function has local scope, while the `scope` variable defined in the main program is a global variable. Even though the two variables have the same name, their scopes are different, and their values are different when accessed within the area of the program where they are defined.

All of this behavior is normal and expected. However, it can all change if you leave off the keyword `var` in the variable definitions. JavaScript allows you to define variables without using the `var` keyword, but when you do, that variable automatically has global scope, even if defined within a function.

Example 1-11 demonstrates the ramifications of leaving off the `var` keyword when defining variables.

Example 1-11. The ramification of overusing global variables

```
function showScope() {
   scope = "local";
   return scope;
}

scope = "global";
print(scope); // displays "global"
print(showScope()); // displays "local"
print(scope); // displays "local"
```

In Example 1-11, because the `scope` variable inside the function is not declared with the `var` keyword, when the string `"local"` is assigned to the variable, we are actually changing the value of the `scope` variable in the main program. You should always begin every definition of a variable with the `var` keyword to keep things like this from happening.

Earlier, we mentioned that JavaScript has function scope. This means that JavaScript does not have *block* scope, unlike many other modern programming languages. With block scope, you can declare a variable within a block of code and the variable is not accessible outside of that block, as you typically see with a C++ or Java `for` loop:

```
for (int i = 1; i <=10; ++i) {
   cout << "Hello, world!" << endl;
}
```

Even though JavaScript does not have block scope, we pretend like it does when we write `for` loops in this book:

```
for (var i = 1; i <= 10; ++i ) {
   print("Hello, world!");
}
```

We don't want to be the cause of you picking up bad programming habits.

Recursion

Function calls can be made recursively in JavaScript. The `factorial()` function defined earlier can also be written recursively, like this:

```
function factorial(number) {
   if (number == 1) {
      return number;
   }
   else {
      return number * factorial(number-1);
   }
}

print(factorial(5));
```

When a function is called recursively, the results of the function's computation are temporarily suspended while the recursion is in progress. To demonstrate how this works, here is a diagram for the `factorial()` function when the argument passed to the function is 5:

```
5 * factorial(4)
5 * 4 * factorial(3)
5 * 4 * 3 * factorial(2)
5 * 4 * 3 * 2 * factorial(1)
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

Several of the algorithms discussed in this book use recursion. For the most part, JavaScript is capable of handling fairly deep recursive calls (this is an example of a relatively shallow recursive call); but in one or two situations, an algorithm requires a deeper recursive call than JavaScript can handle and we instead pursue an iterative solution to the algorithm. You should keep in mind that any function that uses recursion can be rewritten in an iterative manner.

Objects and Object-Oriented Programming

The data structures discussed in this book are implemented as objects. JavaScript provides many different ways for creating and using objects. In this section we demonstrate the techniques used in this book for creating objects and for creating and using an object's functions and properties.

Objects are created by defining a constructor function that includes declarations for an object's properties and functions, followed by definitions for the functions. Here is the constructor function for a checking account object:

```
function Checking(amount) {
   this.balance = amount; // property
   this.deposit = deposit; // function
   this.withdraw = withdraw; // function
   this.toString = toString; // function
}
```

The `this` keyword is used to tie each function and property to an object instance. Now let's look at the function definitions for the preceding declarations:

```
function deposit(amount) {
   this.balance += amount;
}

function withdraw(amount) {
   if (amount <= this.balance) {
      this.balance -= amount;
   }
   if (amount > this.balance) {
      print("Insufficient funds");
   }
}

function toString() {
   return "Balance: " + this.balance;
}
```

Again, we have to use the `this` keyword with the `balance` property in order for the interpreter to know which object's `balance` property we are referencing.

Example 1-12 provides the complete definition for the checking object along with a test program.

Example 1-12. Defining and using the `Checking` object

```
function Checking(amount) {
   this.balance = amount;
   this.deposit = deposit;
   this.withdraw = withdraw;
   this.toString = toString;
}

function deposit(amount) {
   this.balance += amount;
}

function withdraw(amount) {
   if (amount <= this.balance) {
      this.balance -= amount;
   }
   if (amount > this.balance) {
      print("Insufficient funds");
   }
```

```
}

function toString() {
   return "Balance: " + this.balance;
}

var account = new Checking(500);
account.deposit(1000);
print(account.toString()); // Balance: 1500
account.withdraw(750);
print(account.toString()); // Balance: 750
account.withdraw(800); // displays "Insufficient funds"
print(account.toString()); // Balance: 750
```

Summary

This chapter provided an overview of the way we use JavaScript throughout the rest of the book. We try to follow a programming style that is common to many programmers who are accustomed to using C-style languages such as C++ and Java. Of course, JavaScript has many conventions that do not follow the rules of those languages, and we certainly point those out (such as the declaration and use of variables) and show you the correct way to use the language. We also follow as many of the good JavaScript programming practices outlined by authors such as John Resig, Douglas Crockford, and others as we can. As responsible programmers, we need to keep in mind that it is just as important that our programs be readable by humans as it is that they be correctly executed by computers.

CHAPTER 2
Arrays

The array is the most common data structure in computer programming. Every programming language includes some form of array. Because arrays are built-in, they are usually very efficient and are considered good choices for many data storage 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 usually 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 altogether.

A JavaScript array is actually a specialized type of JavaScript object, with the indices being 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 just objects, they are not quite as efficient as the arrays of other programming languages.

While JavaScript arrays are, strictly speaking, JavaScript objects, they are specialized objects categorized internally as arrays. The `Array` is one of the recognized JavaScript object types, and as such, there is a set of properties and functions 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 the elements stored in an array. JavaScript 1.5 also includes array functions that allow programmers

to work with arrays using functional programming techniques. We demonstrate all of these techniques in the following sections.

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 manner, you have an array with length of 0. You can verify this by calling the built-in `length` property:

```
print(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];
print(numbers.length); // displays 5
```

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

```
var numbers = new Array();
print(numbers.length); // displays 0
```

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);
print(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);
print(numbers.length); // displays 10
```

Unlike many other programming languages, but common for most scripting languages, JavaScript array elements do not all have to 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()` function, like this:

```
var numbers = 3;
var arr = [7,4,1776];
print(Array.isArray(number)); // displays false
print(Array.isArray(arr)); // displays true
```

We've covered several techniques for creating arrays. As for which function is best, most JavaScript experts recommend using the [] operator, saying it is more efficient than

calling the `Array` constructor (see *JavaScript: The Definitive Guide* [O'Reilly] and *JavaScript: The Good Parts* [O'Reilly]).

Accessing and Writing Array Elements

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

```
var nums = [];
for (var i = 0; i < 100; ++i) {
   nums[i] = i+1;
}
```

Array elements are also 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];
print(sum); // displays 15
```

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

```
var numbers = [1,2,3,5,8,13,21];
var sum = 0;
for (var i = 0; i < numbers.length; ++i) {
   sum += numbers[i];
}
print(sum); // displays 53
```

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. By using the `length` property, which returns the number of elements currently in the array, you can guarantee that your loop processes all array elements.

Creating Arrays from Strings

Arrays can be created as the result of calling the `split()` function on a string. This function 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()` function 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 aggregate operations you can perform on arrays. First, you can assign one array to another array:

```
var nums = [];
for (var i = 0; i < 10; ++i) {
   nums[i] = i+1;
}
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 simply points 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 the 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 with arrays 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 a simple list.

Accessor Functions

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

Searching for a Value

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

```
var names = ["David", "Cynthia", "Raymond", "Clayton", "Jennifer"];
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()` function will always return the position of the first occurrence. A similar function, `lastIndexOf()`, will 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 functions that return string representations of an array: `join()` and `toString()`. Both functions return a string containing 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()` function with an array name, it automatically calls the `toString()` function for that array:

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

Creating New Arrays from Existing Arrays

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

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

```
var cisDept = ["Mike", "Clayton", "Terrill", "Danny", "Jennifer"];
var dmpDept = ["Raymond", "Cynthia", "Bryan"];
var itDiv = cis.concat(dmp);
print(itDiv);
```

```
itDiv = dmp.concat(cisDept);
print(itDiv);
```

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

The `splice()` function creates a new array from the contents of an existing array. The arguments to the function 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 itDiv = ["Mike","Clayton","Terrill","Raymond","Cynthia","Danny","Jennifer"];
var dmpDept = itDiv.splice(3,3);
var cisDept = itDiv;
print(dmpDept); // Raymond,Cynthia,Danny
print(cisDept); // Mike,Clayton,Terrill,Jennifer
```

There are other uses for `splice()` as well, such as modifying an array by adding and/or removing elements. See the Mozilla Developer Network website for more information.

Mutator Functions

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

Adding Elements to an Array

There are two mutator functions for adding elements to an array: `push()` and `unshift()`. The `push()` function 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 using the `length` property to extend an array:

```
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 function, 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 function for adding array elements to the beginning of an array is `unshift()`. Here is how the function 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,1,2);
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 function.

Removing Elements from an Array

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

```
var nums = [1,2,3,4,5,9];
nums.pop();
print(nums); // 1,2,3,4,5
```

Without mutator functions, removing elements from the beginning of an array requires shifting elements toward 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 function we need to remove an element from the beginning of an array is `shift()`. Here is how the function 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. Both `pop()` and `shift()` return the values they remove, so you can collect the values in a variable:

```
var nums = [6,1,2,3,4,5];
var first = nums.shift(); // first gets the value 9
nums.push(first);
print(nums); // 1,2,3,4,5,6
```

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. However, there is one mutator function 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
```

The elements spliced into an array can be any list of items passed to the function, not necessarily a named array of items. For example:

```
var nums = [1,2,3,7,8,9];
nums.splice(3,0,4,5,6);
print(nums);
```

In the preceding example, the arguments `4`, `5`, and `6` represent the list of elements we want to insert into `nums`.

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 functions 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
```

We often need to sort the elements of an array into order. The mutator function 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()` function sorts data lexicographically, assuming the data elements are strings, even though in the preceding example, the elements are numbers. We can make the `sort()` function work correctly for numbers by passing in an ordering function as the first argument to the function, which `sort()` will then use to sort the array elements. This is the function that `sort()` will use when comparing pairs of array elements to determine their correct order.

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; and if the number returned is positive, the left operand is greater than the right operand.

With this in mind, let's rerun the previous small program 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()` function uses the `compare()` function to sort the array elements numerically rather than lexicographically.

Iterator Functions

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

Non–Array-Generating Iterator Functions

The first group of iterator functions we'll discuss do not generate a new array; instead, they either perform an operation on each element of an array or generate a single value from an array.

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

```
function square(num) {
  print(num, num * num);
}

var nums = [1,2,3,4,5,6,7,8,9,10];
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 iterator function, `every()`, applies a Boolean function to an array and returns `true` if the function can return `true` for every element in the array. Here is an example:

```
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("not all numbers are even");
}
```

The program displays:

```
all numbers are even
```

If we change the array to:

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

the program displays:

```
not all numbers are even
```

The `some()` function 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. For 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];
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()` function applies a function to an accumulator and the 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(runningTotal, currentValue) {
   return runningTotal + currentValue;
}

var nums = [1,2,3,4,5,6,7,8,9,10];
var sum = nums.reduce(add);
print(sum); // displays 55
```

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

```
function concat(accumulatedString, item) {
   return accumulatedString + item;
}

var words = ["the ", "quick ","brown ", "fox "];
var sentence = words.reduce(concat);
print(sentence); // displays "the quick brown fox"
```

JavaScript also provides a `reduceRight()` function, which works similarly to `reduce()`, only working from the righthand side of the array to the left, instead of from left to right. The following program uses `reduceRight()` to reverse the elements of an array:

```
function concat(accumulatedString, item) {
   return accumulatedString + item;
}

var words = ["the ", "quick ","brown ", "fox "];
var sentence = words.reduceRight(concat);
print(sentence); // displays "fox brown quick the"
```

Iterator Functions That Return a New Array

There are two iterator functions that return new arrays: `map()` and `filter()`. The `map()` function works like the `forEach()` function, applying a function to each element of an array. The difference between the two functions is that `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 the `words` array. 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()` function. We accomplish this by calling the `join()` function with the empty string as the separator.

The `filter()` function works similarly to `every()`, but instead of returning `true` if all the elements of an array satisfy a Boolean function, the function 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 is 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("All grades: );
print(grades);
print("Passing grades: ");
print(passGrades);
```

This program displays:

```
All grades:
39,43,89,19,46,54,48,5,13,31,27,95,62,64,35,75,79,88,73,74
Passing grades:
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 recieve,percieve,concieve
```

Two-Dimensional and Multidimensional Arrays

JavaScript arrays are only one-dimensional, but you can create multidimensional 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 the array an array as well. 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 *JavaScript:*

The Good Parts (O'Reilly, p. 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 definition:

```
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"
```

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 preceding code fragment 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 for each student's grades by adding each grade on the row to a `total` variable and then dividing `total` by the total number of grades on that row. 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)` function.

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 calculation for each test:

```
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 the rows in the array may have a different number of elements. One row may have three elements, while another row may have five elements, while yet another row may have just one element. 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 two grades, while the second student has three grades, and the third student has four 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 the 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 consist of objects, and all the functions and properties of arrays work with objects.

For example:

```
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);
print("After push: ");
displayPts(points);
points.shift();
print("After 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
After push:
1, 2
3, 5
2, 8
4, 4
12, -3
After shift:
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 used 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 functions for adding a new temperature and 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()` function uses the `push()` function 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 functions. 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 function for adding a grade and a function for displaying the student's grade average.
2. Store a set of words in an array and display the contents both forward and backward.
3. Modify the `weeklyTemps` object in the chapter so that it stores a month's worth of data using a two-dimensional array. Create functions to display the monthly average, a specific week's average, and all the weeks' averages.
4. Create an object that stores individual letters in an array and has a function for displaying the letters as a single word.