# The JavaScript Environment and Programming Model

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

## The JavaScript Environment

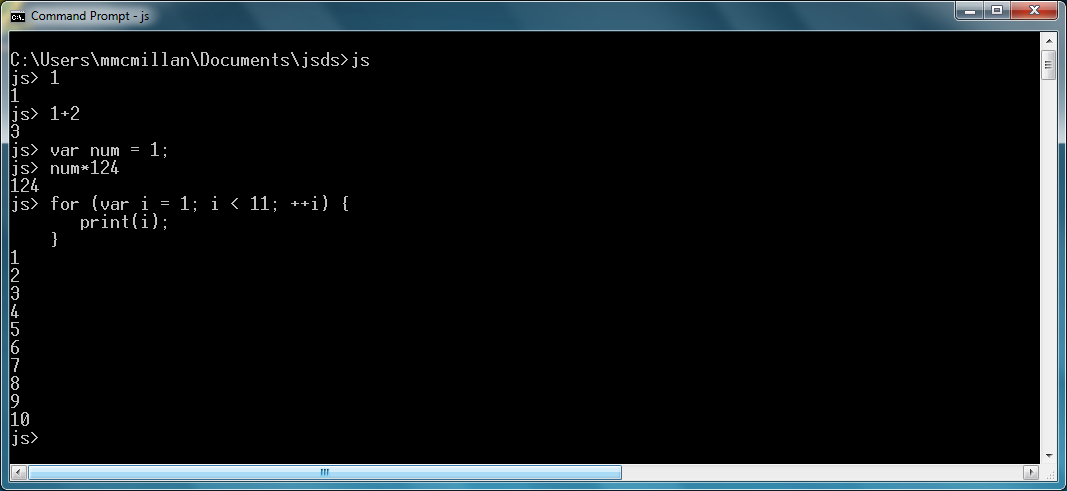
JavaScript has historically been a programming language that ran only inside a web browser. However, in the past few years, there are more and more JavaScript environments that can be run from the desktop or from the server. In this book we use one such environment – the JavaScript shell developed by Mozilla.org.

To get the JavaScript shell, navigate to the following web page:

<http://ftp.mozilla.org/pub/mozilla.org/firefox/nightly/latest-trunk/>

Scroll to the bottom 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 either use it in interactive mode or you can use it to interpret JavaScript programs. To use the shell in interactive mode, type the command, js, from a command prompt. The shell prompt will appear, js> , and you are ready to start entering JavaScript statements. The screen shot below demonstrates how the shell works in interactive mode:



You can type arithmetic expressions and the shell will immediately evaluate them. You can also write any legal JavaScript statement and the shell will immediately evaluate them 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 your JavaScript programs. This is how we will use the shell throughout the rest of the book.

To use the shell to interpret programs, you first have to create a file that contains a JavaScript program. You can use any text editor as long as it can save your 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 name of the file. If we have a file named program.js with the following JavaScript program stored in it:

for (var i = 1; i < 6; ++i) {

print(i);

}

We can execute it by typing the following command at the command prompt:

js program.js

The program will execute and display:

1

2

3

4

5

and return to the command prompt.

(say more stuff about using the shell)

## The JavaScript Programming Model

In this section we discuss how we use JavaScript. We do not consider this to be a tutorial on how to program in JavaScript but just a guide to how we write will present the data structures and algorithms discussed in this book using the JavaScript programming language.

### Declaring and Initializing Variables

JavaScript variables are global by default and, strictly speaking, don't have to be declared before being used. When a JavaScript variable is used without being declared, it becomes a global variable. In this book, however, we follow the convention of modern typed programming languages by declaring all variables before their 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 in JavaScript

JavaScript utilizes the standard arithmetic operators: +, -, \*, /, %, and has a Math library you can use for advanced mathematical 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.

Here are some examples of performing arithmetic in JavaScript as well as examples of using some of the Math functions:

### 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, and the if-else if statement. Here is an example of 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;

}

Here is an example of an 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;

}

Here is an example of an 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. The switch statement provides a more structured, cleaner construction when you have many simple decisions to make. The code below demonstrates how the switch statement works:

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("Either a second-half month or 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 as is necessary in languages such as Java and C++. In fact, you'll notice in the example above 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. Here is an example that sums the integers 1 through 10:

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 set number of times, we use a for loop. Below is an example that sums the elements of an array:

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

var sum = 0;

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

sum += numbers[i];

}

print(sum);

## Functions

JavaScript allows us to define both value-returning functions and functions that do not return values. Both types are just called functions in JavaScript and you do not differentiate between the two types of functions when defining them.

Here is an example of a value-returning function and its use in a program:

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

We can also write functions that don't return values but instead perform operations. Here is an example of a function that modifies the contents of an array:

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 parameters in JavaScript are passed by value and there are no reference parameters in the language. However, there are reference objects, such as arrays, which are passed by reference to functions, as we demonstrated above.

## 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. Here is an example that demonstrates global scope:

function showScope() {

return scope;

}

var scope = "global";

print(scope); // displays "global"

print(showScope()); // displays "global"

The function showScope() can access the variable scope because it is a global variable.

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 remains a global variable, so even though the two variables have the same name, their scopes are different so 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 we leave off the keyword var in the variable definitions. JavaScript allows you to define variables without using the keyword var, but when you do, that variable automatically has global scope, even if defined within a function. The following program demonstrates the ramifications of leaving off the var keyword:

function showScope() {

scope = "local";

return scope;

}

scope = "global";

print(scope); // displays "global"

print(showScope()); // displays "local"

print(scope); // displays "local"

In this example, because the scope variable inside the function is not declared with the var keyword, when we assign scope the value "local" in the function, we are actually changing the value of the scope variable from the main program. You should always begin EVERY definition of a variable with the keyword var to keep things like this from happening.

Earlier, we mentioned that JavaScript has function scope. This means that JavaScript does not have *block* scope, like many other modern languages. For example, with block scope, you can declare a variable within a block of code and the variable is not accessible outside of that block, such 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:

for (var i = 1; i <= 10; ++i ) {

print("Hello, world!");

}

We do not want to be responsible for you picking up any bad programming habits from this book.

## Recursion

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

function factorial(number) {

if (number == 1) {

return number;

}

else {

return number \* factorial(number-1);

}

}

print(factorial(5));

## Objects and Object-Oriented Programming

Most, if not all, of 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 discuss the techniques used in this book for creating objects and for creating and using an object's methods and properties.

### Creating Objects

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

function Checking(amount) {

this.balance = amount;

this.deposit = deposit;

this.withdraw = withdraw;

this.toString = toString;

}

The this keyword is used to tie each method and property to an object instance. Now let's look at the method definitions:

function deposit(amount) {

this.balance += amount;

}

function withdraw(amount) {

if (amount <= this.balance) {

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.

Here is a program that uses the Checking object:

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

We will not be using inheritance in this book so we won't bother demonstrating how JavaScript implements inheritance.