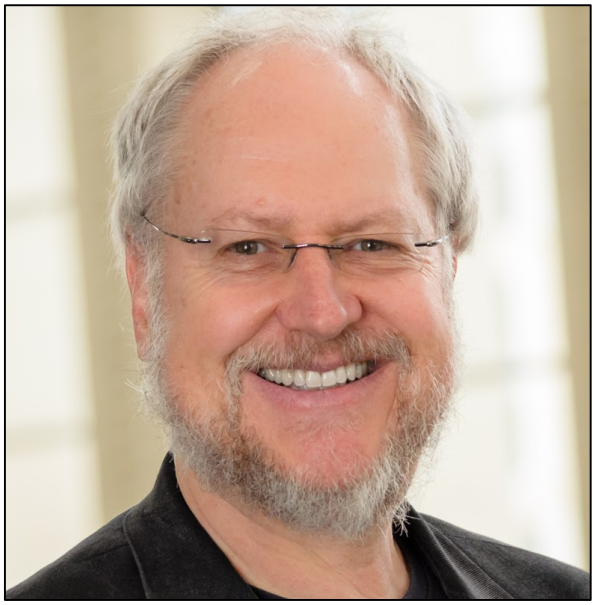
CHAPTER 2
Introducing JavaScript

> Computer programs are the most complex things that humans make.
>
> — Douglas Crockford, *JavaScript: The Good Parts,* 2008



Douglas Crockford (1955–)

Douglas Crockford has written extensively about JavaScript and has for many years championed the virtues of a language that is too often regarded as poorly designed. In his 2008 book *JavaScript: The Good Parts,* Crockford recognizes the negative perceptions of JavaScript but notes that "in JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders." Fortunately, it is possible to write programs that reflect the beauty, elegance, and expressiveness of JavaScript simply by focusing on the good parts of the language and avoiding the pitfalls entirely. The purpose of this book is to teach you only those aspects of the language that support the creation of readable, well-structured programs.

The Karel microworld from Chapter 1 offers a gentle introduction to the idea of programming, but it is missing at least one critically important concept. Although beepers make it possible for Karel to manipulate the contents of its world, Karel offers no effective mechanism for working with data. In computing, the word *data* is usually synonymous with *information.* Computers derive most of their power from their ability to manipulate information in great quantity and at high speed. In most of Europe, computer science is more commonly called ***informatics,*** which emphasizes the central role that information plays.

Before you can appreciate the power of computing, you need to learn at least the basics of a programming language that makes it possible to work with data. The programs in this book use a programming language called ***JavaScript,*** which has become the standard language for writing interactive web applications. The first version of JavaScript appeared in 1995, reportedly written by a single programmer at the Netscape Communications Corporation in just ten days. Because of its popularity, JavaScript is built into every major web browser, which means that any device with a browser can run JavaScript programs without any additional software.

The focus of this book, however, is not on the JavaScript language itself but rather on the programs that you write using that language. This book does not cover all of JavaScript and deliberately avoids those aspects of the language that are easy to misuse. Even so, the subset of JavaScript you will learn in this book gives you the tools you need to write exciting applications that use only the best features of the JavaScript language.

2.1 Data and types

For much of their history, computing machines—even before the age of modern computing—have worked primarily with numeric data. The computers built in the mid 1960s were so closely tied to processing numeric data that they earned the nickname ***number crunchers*** as a result. Information, however, comes in many forms, and computers are increasingly good at working with data of many different types. When you write programs that count or add things up, you are working with ***numeric data.*** When you write programs that manipulate characters—typically assembled into larger units such as words, sentences, and paragraphs—you are working with ***string data.*** You will learn about these and many other data types as you work your way through this book.

In computer science, a data type is defined by two properties: a domain and a set of operations. The ***domain*** is simply the set of values that are elements of that type. For numeric data, the domain consists of numbers like 0, 42, –273, and 3.14159265. For string data, the domain is sequences of characters that appear on the keyboard or that can be displayed on the screen. The ***set of operations*** is the toolbox that

allows you to manipulate values of that type. For numeric data, the set of operations includes addition, subtraction, multiplication, and division, along with a variety of more sophisticated functions. For string data, however, it is hard to imagine what an operation like multiplication might mean. String data offers a different set of operations such as combining two strings to form a longer one or comparing two strings to see if they are in alphabetic order. The general rule is that the set of operations must be appropriate to the elements of the domain. The two components together—the domain and the operations—define a ***data type.***

2.2 Numeric data

Computers today store data in so many exciting forms that numbers may seem a bit boring. Even so, numbers are a good starting point for talking about data, mostly because they are both simple and familiar. You've been using numbers, after all, ever since you learned to count. Moreover, as you'll discover in Chapter 7, all information is represented inside the computer in numeric form.

Representing numbers in JavaScript

One of the important design principles of modern programming languages is that concepts that are familiar to human readers should be expressed in an easily recognizable form. Like most languages, JavaScript adopts that principle for numeric representation, which means that you can write numbers in a JavaScript program in much the same way you would write them anywhere else. Numbers in JavaScript consist of digits, optionally containing a decimal point. Negative numbers are preceded by a minus sign, which is written using a hyphen. Thus, the following examples are all legal JavaScript numbers:

```
0    42    -273    3.14159265    -0.5    1000000
```

Note that large numbers, such as the value of one million shown in the last example, are written without using commas to separate the digits into groups of three.

Numbers can also be written in a special programmer's variant of scientific notation, in which the value is represented as a number multiplied by a power of 10. To write a number using this style, you write a number in standard decimal notation, followed immediately by the letter **E** and an integer exponent, optionally preceded by a **+** or **-** sign. For example, the speed of light in meters per second is approximately

$$2.9979 \times 10^8$$

which can be written in JavaScript as

```
2.9979E+8
```

In JavaScript's scientific notation, the letter **E** stands for the words *times 10 to the power.*

Arithmetic expressions

The real power of numeric data comes from the fact that JavaScript allows you to perform computation by applying mathematical operations to numeric data, ranging in complexity from addition and subtraction up to highly sophisticated mathematical functions. As in mathematics, JavaScript allows you to express those calculations through the use of operators, such as **+** and **-** for addition and subtraction.

As you are learning how JavaScript works, it is useful to have access to some application that allows you to enter JavaScript expressions and see what values they produce. The web site associated with this textbook includes an application that does precisely that, but there are similar facilities available in other JavaScript environments. The examples in this book illustrate interactions with JavaScript in the context of a window called the ***JavaScript console,*** but those examples should be easy to follow even if you are using a different environment.


Tom Lehrer

To get a sense of how interactions with the JavaScript console work, suppose that you want to solve the following problem, which the singer-songwriter, political satirist, and mathematician Tom Lehrer proposed in his song "New Math" in 1965:

$$\begin{array}{r} 342 \\ -173 \\ \hline \end{array}$$

To find the answer, all you have to do is enter the subtraction into the JavaScript console, as follows:

JavaScript Console

```
> 342 - 173
169
>
```

This computation is an example of an ***arithmetic expression,*** which consists of a sequence of values called ***terms*** combined using symbols called ***operators,*** most of which are familiar from elementary-school arithmetic. The arithmetic operators in JavaScript include the following:

+	Addition
-	Subtraction (or negation, if written with no value to its left)
*****	Multiplication
/	Division
%	Remainder

The only one of these operators that may seem unfamiliar is `%`, which computes the remainder of one value divided by another. For example, `7 % 3` has the value 1, because `7 / 3` leaves a remainder of 1. If one number is evenly divisible by another, there is no remainder left over, so that `12 % 4` has the value 0.

Following standard mathematical convention, the multiplication, division, and remainder operations are performed before addition and subtraction, although you can use parentheses to change the evaluation order. For example, if you want to average the numbers 4 and 7, you can enter the following expression on the console:

```
JavaScript Console
> (4 + 7) / 2
5.5
>
```

If you leave out the parentheses, JavaScript first divides 7 by 2 and then adds 4 and 3.5 to produce the value 7.5, as follows:

```
JavaScript Console
> 4 + 7 / 2
7.5
>
```

Parentheses are needed to compute the average.

If JavaScript is your first programming language, the calculation in this example will seem perfectly natural because it follows the conventions of arithmetic that you learned in elementary school. If you have used other languages before, however, JavaScript's treatment of numbers may require you to think about arithmetic expressions in a different way. Most programming languages define two different numeric types: one for whole numbers—which are more commonly referred to as ***integers*** in computer science—and one for numbers with fractional parts. JavaScript has just one numeric type, which makes arithmetic a bit simpler.

The order in which JavaScript evaluates the operators in an expression is governed by their ***precedence,*** which is a measure of how tightly each operator binds to the operands on either side. If two operators compete for the same operand, the one with higher precedence is applied first. If two operators have the same precedence, they are applied in the order specified by their ***associativity,*** which indicates whether that operator groups to the left or to the right. Most operators in JavaScript are ***left-associative,*** which means that the leftmost operator is evaluated first. A few operators, such as the assignment operator discussed later in this chapter, are ***right-associative,*** which means that they group from right to left.

Figure 2-1 shows a complete precedence table for the JavaScript operators, many of which you will have little or no occasion to use. As additional operators are

FIGURE 2-1 Complete precedence table for the JavaScript operators

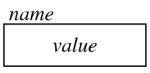
Operators organized into precedence groups	Associativity
`()  []  .`	*left*
unary operators: `-  ++  --  !  ~  typeof  new`	*right*
`*  /  %`	*left*
`+  -`	*left*
`<<  >>  >>>`	*left*
`<  <=  >  >=  instanceof  in`	*left*
`===  !==  ==  !=`	*left*
`&`	*left*
`^`	*left*
`|`	*left*
`&&`	*left*
`||`	*left*
`?:`	*right*
`=  op=`	*right*

introduced later in this book, you can look them up in this table to see where they fit in the precedence hierarchy. Since the purpose of the precedence rules is to ensure that JavaScript expressions obey the same rules as their mathematical counterparts, you can usually rely on your intuition. Moreover, if you are ever in any doubt, you can always include parentheses to make the order of operations explicit.

2.3 Variables

When you write a program that works with data values, it is often convenient to use names to refer to a value that can change as the program runs. In programming, names that refer to values are called ***variables.***

Every variable in JavaScript has two attributes: a *name* and a *value*. To understand the relationship of these attributes, it is best to think of a variable as a box with a label attached to the outside, like this:



The name of the variable appears on the label and is used to tell different boxes apart. If you have three variables in a program, each variable will have a different

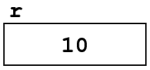
name. The value corresponds to the contents of the box. The name of the box is fixed, but you can change the value as often as you like.

Declaring variables

If you need to create a new variable in JavaScript, the standard approach in modern versions of JavaScript is to include a line in your program that begins with the keyword **`let`** followed by the name of the variable, an equal sign, the initial value for that variable, and finally a semicolon. A program line that introduces a new variable is called a ***declaration.*** The following declaration, for example, introduces a variable named **`r`** and assigns it the value 10:

```
let r = 10;
```

Conceptually, this declaration creates a box inside the computer's memory, gives it the label **`r`**, and stores the value 10 in the box, like this:

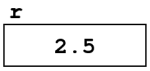


Assignment

Once you have declared a variable, you can change its value by using an ***assignment statement,*** which looks just like a declaration, but without the **`let`** keyword at the beginning. For example, if you execute the assignment statement

```
r = 2.5;
```

the value in the box would change as follows:



The value that appears to the right of the equal sign in either a declaration or an assignment statement can be any JavaScript expression. For example, you can compute the average of the numbers 3, 4, and 5 using the following declaration:

```
let average = (3 + 4 + 5) / 3;
```

Assignment statements are often used to modify the current value of a variable. For example, you could add the value of **`deposit`** to **`balance`** using the statement

```
balance = balance + deposit;
```

which takes the current value of **`balance`**, adds the value of **`deposit`**, and then stores the result back in **`balance`**. Assignment statements of this form are so common that JavaScript allows you to use the following shorthand

```
balance += deposit;
```

Similarly, you can subtract the value of **surcharge** from **balance** by writing

```
balance -= surcharge;
```

More generally, the JavaScript statement

variable op= *expression*;

is equivalent to

variable = *variable op* (*expression*);

where the parentheses are included to emphasize that the entire expression is evaluated before *op* is applied. Such statements are called ***shorthand assignments.***

Increment and decrement operators

Beyond the shorthand assignment operators, JavaScript offers a further level of abbreviation for the particularly common operations of adding or subtracting 1 from a variable. Adding 1 to a variable is called ***incrementing*** it; subtracting 1 is called ***decrementing*** it.

JavaScript indicates these operations in an extremely compact form using the operators **++** and **--**. For example, in JavaScript the statement

```
x++;
```

has the same effect on the variable **x** as

```
x += 1;
```

which is itself short for

```
x = x + 1;
```

Similarly,

```
y--;
```

has the same effect as

```
y -= 1;
```

or

```
y = y - 1;
```

If your curiosity leads you to read JavaScript programs written by experienced programmers, you will quickly discover that the increment and decrement operators are both more complicated and more flexible than these examples suggest. If you need to know these details to understand those programs, you can always read more about these operators on the many web sites that act as reference guides for the language. More often than not, however, writing code that depends on these details gives rise to programs that are difficult to read. To minimize that danger, the programs in this book use `++` and `--` only in their simplest form.

Naming conventions

The names used for variables, constants, functions, and so forth are collectively known as ***identifiers.*** In JavaScript, the rules for identifier formation are

1. The name must start with a letter or the underscore character (_).
2. All other characters in the name must be letters, digits, or the underscore.
3. The name must not be one of the reserved keywords listed in Figure 2-2.

Uppercase and lowercase letters appearing in an identifier are considered to be different. Thus, the identifier `ABC` is not the same as the identifier `abc`.

You can make your programs more readable by using variable names that immediately suggest the meaning of that variable. If `r`, for example, refers to the radius of a circle, that name makes sense because it follows standard mathematical convention. In most cases, however, it is better to use longer names that make it clear to anyone reading your program exactly what value a variable contains. For example, if you need a variable to keep track of the number of pages in a document, it is better to use a name like `numberOfPages` than to use a shorter, more cryptic name like `np`.

FIGURE 2-2 **Reserved words in JavaScript**

`abstract`	`default`	`for`	`new`	`throw`
`arguments`	`delete`	`function`	`null`	`throws`
`await`	`do`	`goto`	`package`	`transient`
`boolean`	`double`	`if`	`private`	`true`
`break`	`else`	`implements`	`protected`	`try`
`byte`	`enum`	`import`	`public`	`typeof`
`case`	`eval`	`in`	`return`	`var`
`catch`	`export`	`instanceof`	`short`	`void`
`char`	`extends`	`int`	`static`	`volatile`
`class`	`false`	`interface`	`super`	`while`
`const`	`final`	`let`	`switch`	`with`
`continue`	`finally`	`long`	`synchronized`	`yield`
`debugger`	`float`	`native`	`this`	

The variable name **numberOfPages** may at first look a little odd because of the capital letters that appear in the middle of the name. That name, however, follows what has become a widely accepted standard for naming variables. By convention, variable names in JavaScript begin with a lowercase letter but include uppercase letters at the beginning of each new word. This convention is called ***camel case*** because it creates uppercase "humps" in the middle of the variable name.

Constants

You can also make your programs more readable by giving names to values that you never expect to change. Such values are called ***constants.*** Modern versions of JavaScript support the declaration of constants simply by replacing the keyword **let** in the declaration with the keyword **const**. For example, if you are writing a program that needs to undertake geometrical calculations involving circles, it is useful to have a constant named **PI** whose value is a reasonable approximation of the mathematical constant π. Although you will discover later in this chapter that the constant **PI** is already defined in one of the standard libraries, you could always define it yourself by writing the following declaration:

```
const PI = 3.14159265;
```

By convention, constant names are written entirely in uppercase using underscores to indicate word boundaries.

Sequential calculations

The ability to define variables and constants makes arithmetic calculations easier to follow, even in the console window. The following sequence of statements, for example, calculates the area of a circle of radius 10:

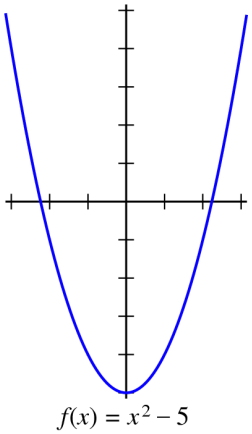
```
JavaScript Console
> const PI = 3.14159265;
> let r = 10;
> let area = PI * r * r;
> area
314.159265
>
```

JavaScript does not include an operator for raising a number to a power, so the easiest way to express the computation of r^2 is simply to multiply r by itself.

2.4 Functions

As you discovered when you wrote simple Karel programs in Chapter 1, you don't need to enter all your computational operations in the console window but can instead store those steps as a function. The big difference between functions in

Karel and JavaScript is that functions can use information supplied by their callers and then give back information in return. The caller sends information to the function by specifying values inside the parentheses that indicate a function call. These values are called ***arguments.*** Inside the function, each of these arguments is assigned to a variable called a ***parameter.*** The function uses these parameters to compute a result, which is delivered back to the caller. This process is called ***returning a result.***



In the context of a programming language like JavaScript, the term *function* is intended to evoke the similar concept in mathematics. A mathematical function like

$$f(x) = x^2 - 5$$

expresses a relationship between the value of x and the value of the function. This relationship is depicted in the graph to the right, which shows how the value of the function changes with respect to the value of x.

Implementing functions in JavaScript

The process of writing functions is best introduced by example. The mathematical function $f(x) = x^2 - 5$ has the following implementation in JavaScript:

```
function f(x) {
   return x * x - 5;
}
```

In this definition, **`x`** is the parameter variable, which is set by the argument passed by the caller. For example, if you were to call **`f(2)`**, the variable **`x`** would be set to the value 2. The **`return`** statement specifies the computation needed to calculate the result. Multiplying **`x`** by itself gives the value 4; subtracting 5 gives the final result of −1, which is passed back to the caller.

When you use the JavaScript application that accompanies this book, you can define this function by typing it into the editing area, just as you did with Karel. Once you have defined the function **`f`**, you can call it from the console like this:

JavaScript Console

```
> f(0)
-5
> f(2)
-1
> f(-3)
4
>
```

Parameter variables and any variables declared inside the body of a function are accessible only from inside that function. For this reason, those variables are called

local variables. By contrast, variables declared outside of any function are ***global variables,*** which can be used anywhere in the program. As programs get larger, using global variables makes those programs more difficult to read and maintain. The programs in this book therefore avoid using any global variables unless they are constants. Thus, a global definition of a constant like **`PI`** is acceptable, but any variable whose value might change will always be declared inside a function.

The ability to define functions and global constants makes it possible to store the steps that calculate the area of a circle, as follows:

```
const PI = 3.14159265;

function circleArea(r) {
   return PI * r * r;
}
```

To call the **`circleArea`** function, all you need to do is specify a value for the radius. For example, given these definitions of **`PI`** and **`circleArea`**, you can then execute the following commands in the console window:

```
JavaScript Console
> circleArea(1)
3.141592653
> circleArea(10)
314.1592653
>
```

You can use functions to compute values that come up in practical situations that are largely outside of traditional mathematics. For example, if you travel outside the United States, you will discover that the rest of the world measures temperatures in Celsius rather than Fahrenheit. The formula to convert a Celsius temperature to its Fahrenheit equivalent is

$$F = \frac{9}{5}C + 32$$

which you can easily translate into the following JavaScript function:

```
function celsiusToFahrenheit(f) {
   return 9 / 5 * c + 32;
}
```

The use of **`celsiusTofahrenheit`** is illustrated in the following sample run:

```
JavaScript Console
> celsiusToFahrenheit(0)
32
> celsiusToFahrenheit(20)
68
>
```

Functions can take more than one argument, in which case both the parameter names in the definition and the argument values in the call are separated by commas. For example, the function

```
const INCHES_PER_FOOT = 12;
const CENTIMETERS_PER_INCH = 2.54;

function feetAndInchesToCentimeters(feet, inches) {
   let totalInches = feet * INCHES_PER_FOOT + inches;
   return totalInches * CENTIMETERS_PER_INCH;
}
```

converts a length specified in feet and inches to the equivalent length in centimeters.

When you call the function **`feetAndInchesToCentimeters`**, you must supply the arguments in the order specified by the parameter list. The first argument specifies the number of feet, and the second specifies the number of inches. The following sample run shows three calls to **`feetAndInchesToCentimeters`**, one showing that one inch is 2.54 centimeters, a second showing that a foot is 30.48 (12×2.54) centimeters, and a third showing that eight feet and four inches (a total of 100 inches) corresponds to a length of 254 centimeters:

JavaScript Console

```
> feetAndInchesToCentimeters(0, 1)
2.54
> feetAndInchesToCentimeters(1, 0)
30.48
> feetAndInchesToCentimeters(8, 4)
254
>
```

Even though a JavaScript function can take more than one argument, a function can return only one result. It is therefore impossible to write a JavaScript function that converts a length in centimeters into two independent values: one of which represents the whole number of feet and one that represents the number of extra inches left over. As you will see later in this chapter and again in Chapter 9, there are several strategies that will allow you to come close to achieving this goal.

Library functions

Like all modern languages, JavaScript predefines certain collections of functions and other useful definitions and makes those collections available to programmers as ***libraries.*** One of the most useful libraries in JavaScript is the **`Math`** library, which includes several mathematical definitions that come up often when you are writing programs, even when those programs don't seem particularly mathematical.

Like most built-in libraries in JavaScript, the **`Math`** library is implemented as part of a ***class,*** which you can think of for the moment simply as a structure that unifies a related set of definitions. Figure 2-3 lists several constants and functions available in the **`Math`** library.

In JavaScript, you can use the facilities available in a class by writing the class name, a dot, and the name of the constant or function you want to use. For example, the expression **`Math.PI`** represents the constant named **`PI`** in the **`Math`**

FIGURE 2-3 **Selected constants and functions from the JavaScript `Math` library**

Mathematical constants

`Math.PI`	The mathematical constant π.
`Math.E`	The mathematical constant *e,* which is the base for natural logarithms.

General mathematical functions

`Math.abs(`*x***`)`**	Returns the absolute value of *x*.
`Math.max(`*x, y, ...***`)`**	Returns the largest of the arguments.
`Math.min(`*x, y, ...***`)`**	Returns the smallest of the arguments.
`Math.sqrt(`*x***`)`**	Returns the square root of *x*.
`Math.round(`*x***`)`**	Returns the closest integer to *x*.
`Math.floor(`*x***`)`**	Returns the largest integer less than or equal to *x*.
`Math.ceil(`*x***`)`**	Returns the smallest integer greater than or equal to *x*.

Logarithmic and exponential functions

`Math.exp(`*x***`)`**	Returns the exponential function of *x* (e^x).
`Math.log(`*x***`)`**	Returns the natural logarithm (base *e*) of *x*.
`Math.log10(`*x***`)`**	Returns the common logarithm (base 10) of *x*.
`Math.pow(`*x, y***`)`**	Returns x^y.

Trigonometric functions

`Math.cos(`*theta***`)`**	Returns the cosine of the radian angle *theta.*
`Math.sin(`*theta***`)`**	Returns the sine of the radian angle *theta.*
`Math.tan(`*theta***`)`**	Returns the tangent of the radian angle *theta.*
`Math.atan(`*x***`)`**	Returns the principal arctangent of *x,* which lies between –π/2 and +π/2.
`Math.atan2(`*y, x***`)`**	Returns the angle between the *x*-axis and the line from the origin to (*x*, *y*).

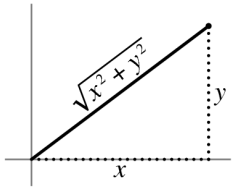
Random number generator

`Math.random()`	Returns a random number that is at least 0 but strictly less than 1.

class, which is defined to be as close an approximation as possible to the mathematical constant π. Similarly, the function call `Math.sqrt(2)` returns the best possible approximation of the square root of 2.

You can use the functions from the `Math` class in writing your own functions. The following function uses the Pythagorean theorem to compute the distance from the origin to the point (x, y):

```
function distance(x, y) {
   return Math.sqrt(x * x + y * y);
}
```



Distance formula

2.5 String data

So far, the programming examples in this chapter have worked only with numeric data. These days, computers work less with numeric data than with string data, which is a generic term for information composed of individual characters. The ability of modern computers to process string data has led to the development of text messaging, electronic mail, word processing systems, social networking, and a wide variety of other useful applications.

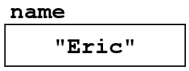
Conceptually, a ***string*** is a sequence of characters taken together as a unit. As in most modern languages, JavaScript includes strings as a built-in type, indicated in a program by enclosing the sequence of characters in quotation marks. For example, the string `"JavaScript"` is a sequence of ten characters including two uppercase letters and eight lowercase letters. The string `"To be, or not to be"` from Hamlet's soliloquy is a sequence of 19 characters including 13 letters, five spaces, and a comma.

JavaScript allows you to use either single or double quotation marks to specify a string, but it is good practice to pick a style and then use it consistently. The programs in this book use double quotation marks, mostly because that convention is common across a wide range of programming languages. The only exception is when the string itself contains a double quotation mark, as in `'"'`, which specifies a one-character string consisting of a double quotation mark. You will learn another way to solve this problem in Chapter 7.

For the most part, you can use strings as a JavaScript data type in much the same way that you use numbers. You can, for example, declare string variables and assign them values, just as you would with numeric variables. For example, the declaration

```
let name = "Eric";
```

declares a variable called **name** and initializes it to the four-character string **"Eric"**. As with the code used earlier in the chapter to declare numeric variables, the easiest way to represent a string-valued variable is to draw a box with the name on the outside and the value on the inside, like this:

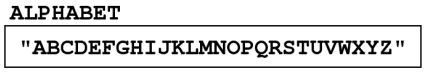


The quotation marks are not part of the string but are nonetheless included in box diagrams to make it easier to see where the string begins and ends.

Similarly, you can declare string constants, as in the following example:

```
const ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

This declaration defines the constant **ALPHABET** to be a string consisting of the 26 uppercase letters, as illustrated by the following box diagram:



String operations

In Section 2.1, you learned that data types are defined by two properties: a *domain* and a *set of operations*. For strings, the domain is the set of all sequences of characters. In JavaScript, most string operations are defined as part of the **String** class, which is covered in detail in Chapter 7. For the moment, it is sufficient to learn just two string operations:

1. Determining the length of a string by adding **.length** to the end of the string. For example, **ALPHABET.length** has the value 26.
2. Joining two strings together end to end, which is called ***concatenation.***

In JavaScript, you indicate concatenation by using the **+** operator, which is the same operator used to indicate addition for numbers. When JavaScript evaluates the **+** operator, it first checks the types of the operands to see which of the two possible interpretations—addition or concatenation—applies. If both operands are numeric, JavaScript chooses addition. If, however, either or both of the operands are strings, JavaScript interprets the **+** operator as concatenation. For example, the expression

```
2 + 2
```

has the value 4, because both of the operands to **+** are numbers. Conversely,

```
"hello" + "world"
```

produces the ten-character string **"helloworld"**.

In this example, it is important to observe that the concatenation operator does not introduce a space character or any other separator between the words. If you want to combine two strings into a single string that represents two distinct words, you have to include the space explicitly. For example, the expression

```
"hello" + " " + "world"
```

combines the three strings to produce the eleven-character string `"hello world"`.

The concatenation operator also allows you to combine string data with other data types. If one of the operands to `+` is a string but the other is some other value, JavaScript automatically converts that value to a string before performing the concatenation. For example, the expression

```
"Fahrenheit " + 451
```

produces the string `"Fahrenheit 451"` because JavaScript converts the numeric value 451 to the string `"451"` before combining the strings together.

Writing simple string functions

Although you will need the additional operations from Chapter 7 to write anything more than the simplest functions, it is worth looking at a few examples that use only the concatenation operator. The following function

```
function doubleString(str) {
   return str + str;
}
```

returns two copies of the supplied string joined together. This function enables the following sample run:

JavaScript Console

```
> doubleString("a")
aa
> doubleString("boo")
booboo
> doubleString("hots")
hotshots
>
```

You can also use concatenation to provide a partial solution to the problem raised earlier in the chapter of converting a distance in centimeters to the equivalent distance in feet and inches. Although JavaScript does not allow you to return two separate values from a function, you can display the correct answer by returning a string that contains both of the desired values, as illustrated by the following function that makes use of the same constants introduced earlier in the chapter:

```
function centimetersToFeetAndInches(cm) {
   let totalInches = cm / CENTIMETERS_PER_INCH;
   let feet = Math.floor(totalInches / INCHES_PER_FOOT);
   let inches = totalInches % INCHES_PER_FOOT;
   return feet + "ft " + inches + "in";
}
```

The following sample run shows three calls to **`centimetersToFeetAndInches`**, one for each of the values produced earlier by **`feetAndInchesToCentimeters`**:

JavaScript Console

```
> centimetersToFeetAndInches(2.54)
0ft 1in
> centimetersToFeetAndInches(30.48)
1ft 0in
> centimetersToFeetAndInches(254)
8ft 4in
>
```

Summary

In this chapter, you have started your journey toward programming in JavaScript by seeing several example programs that make use of two different data types: numbers and strings.

Important points introduced in the chapter include:

- The focus of this book is not on the JavaScript language itself but instead on the principles you need to understand the fundamentals of programming. To reduce the number of language details you need to master, this text uses only those features of JavaScript that Douglas Crockford, whose contributions are described at the beginning of the chapter, identifies as the "good parts" of the language.
- Data values come in many different types, each of which is defined by a *domain* and a *set of operations*.
- Numbers in JavaScript are written in conventional decimal notation. JavaScript also allows numbers to be written in scientific notation by adding the letter **`E`** and an exponent indicating the power of 10 by which the number is multiplied.
- Expressions consist of individual *terms* connected by *operators*. The subexpressions to which an operator applies are called its *operands*.
- The order of operations is determined by *rules of precedence*. The complete table of operators and their precedence appears in Figure 2-1 on page 44.
- *Variables* in JavaScript have two attributes: a name and a value. Variables used in a JavaScript program are *declared* using a line of the form

```
let identifier = expression;
```

which establishes the name and initial value of the variable.

- *Constants* are used to specify values that do not change within a program. You can declare constants in JavaScript by replacing the keyword **let** with the keyword **const** in a declaration. By convention, the names of constants are written entirely in upper case, using the underscore to mark word boundaries.
- You can change the value of variables through the use of *assignment statements*. When you assign a new value to a variable, any previous value is lost.
- JavaScript includes an abbreviated form of the assignment statement in which the statement

  ```
  variable op= expression;
  ```

  acts as a shorthand for the longer expression

  ```
  variable = variable op (expression);
  ```

- A *function* is a block of code that has been organized into a separate unit and given a name. Other parts of the program can then *call* that function, possibly passing it information in the form of *arguments* and receiving a result *returned* by that function.
- Variables declared inside the body of a function are called *local variables* and are visible only inside that function. Variables declared outside of any function are *global variables,* which can be used anywhere in the program. Because using global variables makes programs more difficult to read and maintain, this book avoids using them except for constant definitions.
- A function that returns a value must have a **return** statement that specifies the result. Functions may return values of any type.
- JavaScript's **Math** library defines a variety of functions that implement such standard mathematical functions as **sqrt**, **sin**, and **cos**. A list of the more common mathematical functions appears in Figure 2-3 on page 52.
- A *string* is a sequence of characters taken together as a unit. In JavaScript, you write a string by enclosing its characters in quotation marks. JavaScript accepts either single or double quotation marks for this purpose; this book uses double quotation marks to maintain a consistent convention.
- Although strings support many additional operations that will be presented in Chapter 7, the examples in this chapter and the next few chapters use only the **length** field and the **+** operator. If both operands to **+** are numeric, it is interpreted as addition; if either operand is a string, the string representations of both operands are *concatenated* together end to end.

Review questions

1. What are the two attributes that define a data type?

2. Identify which of the following are legal numbers in JavaScript:

   a) `42`
   b) `-17`
   c) `2+3`
   d) `-2.3`
   e) `20`
   f) `2.0`
   g) `1,000,000`
   h) `3.1415926`
   i) `123456789`
   j) `0.000001`
   k) `1.1E+11`
   l) `1.1X+11`

3. Rewrite the following numbers using JavaScript's form for scientific notation:

   a) 6.02252×10^{23}
   b) 29979250000.0
   c) 0.00000000529167
   d) 3.1415926535

   By the way, each of these values is an approximation of an important scientific or mathematical constant: (a) Avogadro's number, which is the number of molecules in one mole of a chemical substance (b) the speed of light in centimeters per second, (c) the Bohr radius in centimeters, which is the average radius of an electron's orbit around a hydrogen atom in its lowest-energy state, and (d) the mathematical constant π. In the case of π, there is no advantage in using the scientific notation form, but it is nonetheless possible.

4. Indicate which of the following are legal variable names in JavaScript:

   a) `x`
   b) `formula1`
   c) `average_rainfall`
   d) `%correct`
   e) `short`
   f) `tiny`
   g) `total output`
   h) `aReasonablyLongVariableName`
   i) `12MonthTotal`
   j) `marginal-cost`
   k) `b4hand`
   l) `_stk_depth`

5. What does the `%` operator signify in JavaScript?

6. True or false: The `-` operator has the same precedence when it is used before an operand to indicate negation as it does when it is used to indicate subtraction.

7. By applying the appropriate precedence rules, calculate the result of each of the following expressions:

   a) `6 + 5 / 4 - 3`
   b) `2 + 2 * (2 * 2 - 2) % 2 / 2`
   c) `10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1`
   d) `1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) - 9) - 10`

8. What shorthand assignment statement would you use to multiply the value of the variable `salary` by 2?

9. What is the most common way in JavaScript to write a statement that has the same effect as the statement

```
x = x + 1;
```

10. What syntactic form does JavaScript use to refer to a constant or a function in its mathematical library?

11. What is the value of each of the following expressions:

    a) `Math.round(5.99)`
    b) `Math.floor(5.99)`
    c) `Math.ceil(5.99)`
    d) `Math.floor(-5.99)`
    e) `Math.sqrt(Math.pow(3, 2) + Math.pow(4, 2))`

12. What is the possible range of values returned by the function `Math.random`?

13. How do you specify a string value in JavaScript?

14. If a string value is stored in the variable `str`, how would you determine its length?

15. What is meant by the term *concatenation?*

16. How does JavaScript know whether to interpret the `+` operator as addition or concatenation?

17. Given the definition of the `doubleString` function on page 55, what value does JavaScript produce if you call `doubleString(2)`? In light of this behavior, would it be reasonable to shorten the name of the function to `double`? Why or why not?

19. Evaluate each of the following expressions:

    a) `123 + 456`
    b) `123 + "456"`
    c) `"Catch-" + 2 + 2`
    d) `"Citizen" + 2 * 2`

Exercises

1. How would you implement the following mathematical function in JavaScript:

$$f(x) = x^2 - 5x + 6$$


Carl Friedrich Gauss

2. As mathematical historians have told the story, the German mathematician Carl Friedrich Gauss (1777–1855) began to show his mathematical talent at a very early age. When he was in primary school, Gauss was asked by his teacher to compute the sum of the first 100 integers. Gauss is said to have given the answer instantly by working out that the sum of the first *N* integers is given by the formula

$$\frac{N \times (N + 1)}{2}$$

Write a function **`sumFirstNIntegers`** that takes the value of *N* as its argument and returns the sum of those integers, as illustrated in the following sample run:

SumFirstNIntegers

```
> sumFirstNIntegers(3)
6
> sumFirstNIntegers(100)
5050
>
```

3. Write a function **`quotient`** that takes two numbers, **`x`** and **`y`** (which you may assume are both positive integers), and returns the integral quotient of **`x / y`**, discarding any remainder. For example, calling **`quotient(9, 4)`** should return 2 because four goes into nine twice with a remainder of one left over. This function is easy to write if you use the **`Math.floor`** function; the challenge in this exercise is to write **`quotient`** using only the standard arithmetic operators.

4. There's an old nursery rhyme that goes like this:

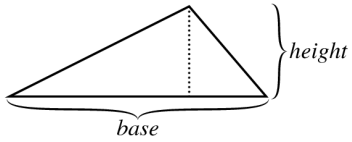
> As I was going to St. Ives,
> I met a man with seven wives,
> Each wife had seven sacks,
> Each sack had seven cats,
> Each cat had seven kits:
> Kits, cats, sacks, and wives,
> How many were going to St. Ives?

The last line turns out to be a trick question: only the speaker is going *to* St. Ives; everyone else is presumably heading in the opposite direction. Suppose, however, that you want to find out how many representatives of the assembled multitude—kits, cats, sacks, and wives—were coming *from* St. Ives. Write a function that takes no arguments and calculates this result. Try to make your function follow the structure of the problem so that anyone reading your code would understand what value it is calculating.

5. Using the **`celsiusToFahrenheit`** function on page 50 as a model, write the function **`fahrenheitToCelsius`** that converts a temperature value in the opposite direction. The conversion formula is

$$C = \frac{5}{9}(F - 32)$$

6. Write a function that computes the area of a triangle given values for its base and its height as illustrated in the following diagram:



Given any triangle, the area is always one half of the base times the height.

7. Write a function that computes the volume of a sphere from its radius using the formula

$$V = \frac{4}{3}\pi r^3$$

8. Write a function **`quote`** that takes a string value and adds double quotation marks at both the beginning of the end. Your function definition should allow you to replicate the following sample run:

Quote

```
> quote("hello")
"hello"
> quote("Fahrenheit " + 11 * 41)
"Fahrenheit 451"
> "   "

> quote("   ")
"   "
>
```

As the lines at the end of this example indicate, the **`quote`** function can make it easier to see where a string begins and ends, particularly if the string contains spaces.

9. *It is a beautiful thing, the destruction of words.*

—Syme in George Orwell's *1984*

In Orwell's novel, Syme and his colleagues at the Ministry of Truth are engaged in simplifying English into a more regular language called *Newspeak.* As Orwell describes in his appendix entitled "The Principles of Newspeak," words can take a variety of prefixes to eliminate the need for the massive number of words we have in English. For example, Orwell writes

> Any word—this again applied in principle to every word in the language—could be negatived by adding the affix *un-*, or could be strengthened by the affix *plus-*, or, for still greater emphasis, *doubleplus-*. Thus, for example, *uncold* meant "warm," while *pluscold* and *doublepluscold* meant, respectively, "very cold" and "superlatively cold."

Define three functions—**`negate`**, **`intensify`**, and **`reinforce`**—that take a string and add the prefixes **`"un"`**, **`"plus"`**, and **`"double"`** to that string, respectively. Your function definitions should allow you to generate the following console session:

Newspeak

```
> negate("cold")
uncold
> intensify("cold")
pluscold
> reinforce(intensify("cold"))
doublepluscold
> reinforce(intensify(negate("good")))
doubleplusungood
>
```