Foreword

Along with many of my colleagues, I have become convinced that JavaScript has significant advantages over Java as a language for the introductory programming course. In purely practical terms, JavaScript is far and away the most successful language in terms of its integration with web and has recently become the most widely used language in the industry overall. At the same time, JavaScript has significant pedagogical advantages over Java. It is much simpler to use and offers a substantially cleaner implementation of several concepts that are essential to modern computer science.

In spite of those advantages, JavaScript has its critics. As the noted JavaScript expert Douglas Crockford concedes, “JavaScript is a language with more than its share of bad parts.” But that is by no means the whole of the story. In the same essay, Crockford goes on to offer the following observation:

Fortunately, JavaScript has some extraordinarily good parts. In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy. My intention here is to expose the goodness in JavaScript, an outstanding, dynamic programming language.

We are convinced that it is possible to teach JavaScript in a way that foregrounds the “beautiful, elegant, highly expressive language” and jettisons almost all of its ugly parts, to a far greater extent than is possible in Java.

Jerry Cain and I taught a pilot version of a JavaScript-based CS 106A in the spring of 2016-17. We learned a great deal from that exercise and have integrated those insights and understandings into this revision of the reader. I want to express my profound gratitude to Jerry who, in addition to being a wonderful colleague in the pilot offering of the course, has been a tireless reader of this text through many drafts. His comments have been amazingly thorough and have made the text far stronger.

I am also indebted to last spring’s section leaders and students who discovered many errors and identified various sources of confusion. Their feedback and our own experience have led to changes in both the reader and the course. Those changes will, in turn, give you an even more exciting start on your journey into the wonders of computer science.

Eric Roberts
Stanford University
September 2017
# Contents

1 A Gentle Introduction 1
   1.1 Introducing Karel 2
   1.2 Teaching Karel to solve problems 4
   1.3 Control statements 10
   1.4 Stepwise refinement 18
   1.5 Algorithms in Karel’s world 21
       Summary 24
       Review questions 25
       Exercises 26

2 Introducing JavaScript 39
   2.1 Data and types 40
   2.2 Numeric data 41
   2.3 Variables 44
   2.4 Functions 48
   2.5 String data 53
       Summary 56
       Review questions 58
       Exercises 59

3 Running Programs in the Browser 63
   3.1 The “Hello World” program 64
   3.2 Introducing the graphics library 68
   3.3 Classes in the graphics library 73
   3.4 Class hierarchies 84
   3.5 Functions that return graphical objects 87
   3.6 Testing and debugging 88
       Summary 95
       Review questions 98
       Exercises 100
4 Control Statements 103
  4.1 Boolean data 104
  4.2 The if statement 109
  4.3 The switch statement 112
  4.4 The while statement 114
  4.5 The for statement 118
  4.6 Algorithmic programming 126
  4.7 Avoiding fuzzy standards of truth 134
      Summary 135
      Review questions 136
      Exercises 137

5 Functions 143
  5.1 A quick review of functions 144
  5.2 Libraries 148
  5.3 The mechanics of function calls 158
  5.4 Decomposition 165
      Summary 172
      Review questions 173
      Exercises 174

6 Writing Interactive Programs 181
  6.1 First-class functions 182
  6.2 A simple interactive example 185
  6.3 Controlling properties of objects 187
  6.4 Responding to mouse events 189
  6.5 Timer-based animation 194
  6.6 Additional GObject subclasses 200
      Summary 209
      Review questions 210
      Exercises 211
7 Strings
7.1 Binary representation 220
7.2 String operations 227
7.3 Common string patterns 232
7.4 String applications 234
7.5 Reading from the console 242
    Summary 244
    Review questions 245
    Exercises 246

8 Arrays
8.1 Introduction to arrays 256
8.2 Reading files 265
8.3 Using arrays for tabulation 271
8.4 Multidimensional arrays 275
8.5 Image processing 276
    Summary 286
    Review questions 287
    Exercises 288

9 Objects
9.1 Objects in JavaScript 296
9.2 Defining a type to represent points 298
9.3 Libraries and interfaces 303
9.4 Adding methods to objects 305
9.5 Implementing inheritance 308
9.6 Using objects as maps 314
9.7 Data-driven programs 316
    Summary 326
    Review questions 326
    Exercises 327

Index 335
In many schools today . . . the computer is being used to program the child. In my vision, the child programs the computer and, in doing so, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model-building.


In the 1960s, Professor Seymour Papert at MIT used a language called LOGO to teach programming to schoolchildren in the Boston area, who wrote programs to control a robotic turtle. The turtle could move forward or backward, rotate a specified number of degrees around its center, and draw pictures on large sheets of paper with a pen mounted on its underside. The LOGO turtle thereby became the first programming microworld, designed to teach the basics of computation in a simplified environment.
This book is about the ideas that underlie the science of computing. You won’t, however, get much out of it through reading alone. Computing, after all, is an activity. As with most activities, one learns computing best through practice. To get you started, this book provides you with the tools you need to solve simple computational problems on your own. That process of necessity involves *programming*, which is the process of transforming a strategy for solving a problem into a precise formulation that can be executed by a computer.

At the same time, it is hard to learn programming through the metaphorical equivalent of jumping in at the deep end of the pool. You have to approach the subject more gradually. Modern programming languages involve so many details that their complexity gets in the way of understanding the bigger picture.

To avoid overwhelming beginners with the intricacies inherent in those languages, computer science courses often introduce programming in the context of a simplified environment called a *microworld*. By design, microworlds are easy to learn and enable students to start programming right away. In the process, those students become familiar with the fundamental concepts of programming without having to master a lot of extraneous details.

Many different microworlds have flourished over the years, including the Project LOGO Turtle described briefly on the title page of this chapter. This book uses a microworld called Karel that we have used with great success in our introductory courses here at Stanford for more than 30 years. Using Karel enables you to solve challenging problems from the very beginning. And because the Karel environment encourages imagination and creativity, you can have a lot of fun along the way.

### 1.1 Introducing Karel

In the 1970s, a Stanford graduate student named Rich Pattis decided that it would be easier to teach the fundamentals of programming if students could learn those ideas in an environment free from the complexities that characterize most programming languages. Drawing inspiration from the success of the LOGO project, Pattis designed a microworld in which students teach a virtual robot to solve simple problems. Pattis called his robot *Karel* after the Czech playwright Karel Čapek whose 1923 play *R.U.R. (Rossum’s Universal Robots)* gave the word *robot* to the English language. Karel was an immediate success and soon spread to universities all over the world.

**Programming in Karel**

Karel is a very simple robot living in a very simple world. By giving Karel a set of instructions, you can direct it to perform certain tasks within its world. Those
instructions constitute a program. Generically, the text that makes up a program is called code. When you write a Karel program, you must do so in a precise way so that Karel can correctly interpret it. Every program you write must obey a set of syntactic rules that define whether that program is legal.

In many respects, the rules of the Karel programming language are similar to those you will see in more sophisticated languages. The most important difference is that Karel’s programming language is tiny—so small, in fact, that it is easy to learn everything there is to know about the Karel language in less than an hour. The details are easy to master. Even so, you will discover that solving a problem in Karel’s world can be extremely challenging. Solving problems is the essence of programming. You learn the rules to unlock the problem-solving power.

Karel’s world

Karel’s world is defined by streets running from west to east and avenues running from south to north. The intersection of a street and an avenue is called a corner. Karel can only be positioned on corners and must be facing in one of the four standard compass directions (north, east, south, and west). In the following sample world, Karel is facing east at the corner of 1st Street and 1st Avenue:

Several other components of Karel’s world can be seen in this example. The object in front of Karel is a beeper. According to Rich Pattis, beepers are “plastic cones which emit a quiet beeping noise.” Karel can only detect a beeper if it is on the same corner. The solid lines in the diagram are walls. Karel’s world always has walls along the edges, but the world may also contain internal walls that serve as barriers.

Karel’s built-in functions

The operations that Karel performs as it executes a program are called functions. When Karel is shipped from the factory, it knows how to execute only the four functions shown in Figure 1-1. The parentheses that appear in each of these examples are part of Karel’s syntax and specify that you want to perform that operation, which in programming terminology is known as calling the function.

Several of the built-in functions place specific restrictions on Karel’s activities. If Karel tries to do something illegal, such as moving through a wall or picking up a
nonexistent beeper, an *error condition* occurs. Whenever an error arises, Karel displays a message describing what went wrong and stops executing the program.

### 1.2 Teaching Karel to solve problems

For the most part, learning to program in Karel is a matter of figuring out how to use Karel’s limited set of operations to solve a specified problem. As a simple example, suppose that you want Karel to move the beeper from its initial position on 2nd Avenue and 1st Street to the center of the ledge at 5th Avenue and 2nd Street. Thus, your goal is to write a Karel program that accomplishes the task illustrated in the following before-and-after diagram:

#### Getting started

The first few steps in solving this problem are simple enough. You need to tell Karel to move forward, pick up the beeper, and then move forward again to reach the base of the ledge. The Karel simulator allows you to execute instructions by typing them into an interactive window called the *Karel console*. The first three steps in the program therefore look like this:
1.2 Teaching Karel to solve problems

Executing these function calls leaves Karel in the following position:

```
+ + + + +
+ + + + +
+ + + + +
```

From here, Karel’s next step is to turn left to begin climbing the ledge. That operation is also easy, because Karel’s set of built-in functions includes \texttt{turnLeft}. Calling \texttt{turnLeft} at the end of the preceding program leaves Karel facing north on the corner of 3rd Avenue and 1st Street. If you then call the \texttt{move} instruction, Karel will move north to reach the following position:

```
+ + + + +
+ + + + +
+ + + + +
```

The next thing you need to do is get Karel to turn right so that it is again facing east. While this operation is conceptually as easy as getting Karel to turn left, there is a slight problem: Karel’s language includes a \texttt{turnLeft} instruction, but no \texttt{turnRight} instruction. It’s as if you bought the economy model only to discover that it is missing an important feature.

At this point, you have your first opportunity to begin thinking like a programmer. You have access to a set of Karel functions, but not exactly the set you need. What can you do? Can you accomplish the effect of a \texttt{turnRight} function using only the capabilities you have? The answer, of course, is yes. You can turn right by turning left three times. After three left turns, Karel will be facing in the desired direction. The next three steps in the program might therefore be

```
\texttt{turnLeft()}
\texttt{turnLeft()}
\texttt{turnLeft()}
```

Although turning left three times has the desired effect, it is hardly an elegant solution. What you as the programmer want to say is

```
\texttt{turnRight()}
```

The only difficulty is that Karel doesn’t yet have a definition for the \texttt{turnRight} function. To use this more expressive operation in your program, you first have to teach Karel what \texttt{turnRight} means.
**Defining functions**

One of the most powerful features of the Karel programming language is the ability to define new functions. Whenever you have a sequence of Karel operations that performs some useful task—such as turning right—you can give that sequence a name. The operation of encapsulating a sequence of instructions under a new name is called *defining a function*. The format for defining a function looks like this:

```
function name() {
    statements that make up the body of the function
}
```

In this pattern, `name` represents the name you have chosen for the new function. To complete the definition, all you have to do is specify the statements between the curly braces. The only difference between the statements in a function and those you enter on the console is that each statement in a Karel function must end with a semicolon. For example, you can use the editor window to define the `turnRight` function as follows, usually as part of a larger program:

```
function turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
}
```

Once you’ve defined a function like `turnRight`, you can think of it as a new primitive operation, just like `move` or `turnLeft`. In a sense, defining a new function is like buying an upgrade for your robot that includes the missing operations.

**Completing the program**

After turning right to face the top of the ledge, the rest of the program is easy. All you need to do is move forward twice, put down the beeper, and then move forward to reach the desired final state. The complete sequence of Karel operations you need to solve the program from beginning to end looks like this:

```
> move()
> pickBeeper()
> move()
> turnLeft()
> move()
> turnRight()
> move()
> move()
> putBeeper()
> move()
```
Instead of typing each instruction into the console, it makes sense to define a new function that contains this sequence of instructions. You can then call that function with a single name. That function, which appears in Figure 1-2 together with the definition of `turnRight`, constitutes a complete Karel program.

In addition to the definitions of the functions `moveBeeperToLedge` and `turnRight`, Figure 1-2 also includes two examples of an important programming feature called a comment, which consists of text designed to explain the operation of the program to human readers. In Karel, comments begin with the characters `/*` and end with the characters `*/`. The first comment describes the operation of the program as a whole; the second describes the `turnRight` function. In a program this short, such comments may seem unnecessary. As programs become more complicated, however, comments quickly become essential tools to document the program design and make it easier for other programmers to understand.

**Using library functions**

Although the code in Figure 1-2 explicitly includes the definition of `turnRight`, it is tedious to have to copy that code into every program that needs that function. For
the most common operations, it makes sense to store them in a way that makes it easy to reuse them in other programs. In computer science, collections of useful functions and other program components are called libraries. For example, the \texttt{turnRight} function and the equally useful \texttt{turnAround} function are both included in a library called \texttt{turns}, which you can use simply by including the following line at the beginning of your program:

\begin{verbatim}
import "turns";
\end{verbatim}

This statement imports the \texttt{turns} library, which includes definitions for the functions \texttt{turnRight} and \texttt{turnAround}. All the Karel examples in the rest of this chapter make use of this feature.

\section*{Decomposition}

Whenever you begin the solution of a programming problem—no matter whether that program is written in Karel or a more advanced programming language—your first task is to figure out how to divide the complete problem into smaller pieces called subproblems, each of which can be implemented as a separate function. That process is called decomposition. Decomposition is one of the most powerful strategies that programmers use to manage complexity, and you will see it again and again throughout this book.

To get a sense of how decomposition works in the context of a very simple problem, imagine that Karel is standing on a “road” as shown on the left side of the following before-and-after diagram:

Karel’s job is to fill each of the two potholes—the one on 2\textsuperscript{nd} Avenue and the one on 5\textsuperscript{th} Avenue—with a beeper and then continue on to the next corner, ending up in the position shown on the right.

Although you could solve this problem using the four predefined instructions, you can use functions to improve the structure of your program. If nothing else, you can use \texttt{turnAround} and \texttt{turnRight} to shorten the program and make its intent clearer. More importantly, you can use decomposition to break the problem down into subproblems and then solve those problems independently. You can, for example, break the problem of filling the pothole into the following subproblems:
1. Move one block forward to reach the first pothole on 2nd Avenue.
2. Fill the pothole by dropping a beeper into it.
3. Move three blocks forward to reach the second pothole on 5th Avenue.
4. Fill the pothole by dropping a beeper into it.
5. Move one block forward to reach the desired final position.

If you think about the problem in this way, you can use functions to ensure that the program reflects your conception of the problem structure, as shown in Figure 1-3.

```plaintext
/*
* File: FillTwoPotholes.k
* ________________
* This program instructs Karel to fill two potholes.
*/
import "turns";

/*
* This function fills two potholes, which must be on 2nd and 5th Avenues.
*/
function fillTwoPotholes() {
    move();
    fillPothole();
    move();
    move();
    move();
    fillPothole();
    move();
}

/*
* Fills a pothole immediately underneath Karel. When you call
* this function, Karel must be standing just above the pothole,
* facing east. When the function returns, Karel will be in its
* original position above the repaired pothole.
*/
function fillPothole() {
    turnRight();
    move();
    putBeeper();
    turnAround();
    move();
    turnRight();
}
As with any programming problem, there are other decomposition strategies you might have tried. Some strategies make the program easier to read, while others only make the meaning more opaque. As programming problems become more complex, decomposition will turn out to be one of the most important aspects of the design process.

Choosing an effective decomposition is much more of an art than a science, although you will find that you get better with practice. Section 1.4 presents some general guidelines that will help you in that process.

### 1.3 Control statements

As useful as it is, the ability to define new functions does not actually enable Karel to solve any new problems. Because each function name is merely shorthand for a specific set of instructions, it is always possible to expand a program written as a series of function calls into a single function that accomplishes the same task, although the resulting code is likely to be long and difficult to read. The instructions are still executed in a fixed order that does not depend on the state of Karel’s world. Before you can solve more interesting problems, you need to learn how to write programs in which this strictly linear, step-by-step order of operations does not apply. To unlock the extraordinary power that this ability provides, you need to learn several new statements in Karel’s programming language that enable Karel to examine its world and change its execution pattern accordingly.

Statements that affect the order in which a program executes instructions are called **control statements**. Control statements fall into the following two classes:

1. **Conditional statements.** Conditional statements specify that certain statements in a program should be executed only if a particular condition holds. In Karel, you specify conditional execution using an **if** statement.

2. **Iterative statements.** Iterative statements specify that certain statements in a program should be executed repeatedly, forming what programmers call a **loop.** Karel supports two iterative statements: a **repeat** statement that allows you to repeat a set of instructions a fixed number of times, and a **while** statement that allows you to repeat a set of instructions as long as some condition holds.

**Conditional statements**

To get a sense of where conditional statements might come in handy, let’s go back to the pothole-filling program presented at the end of section 1.2. Before filling the pothole in the **fillPothole** function, Karel might want to check to see if some other repair crew has already filled the hole, which means that there is already a beeper on that corner. If so, Karel does not need to put down a second one. To
represent such checks in the context of a program, you need to use the `if` statement, which has one of the following two forms:

\[
\text{if (conditional test) } \{
\text{statements to be executed only if the condition is true}
\}
\]

or

\[
\text{if (conditional test) } \{
\text{statements to be executed if the condition is true}
\} \text{ else } \{
\text{statements to be executed if the condition is false}
\}
\]

The first form of the `if` statement is useful when you want to perform an action only under certain conditions. The second is appropriate when you need to choose between two alternative courses of action.

The conditional test shown in the first line of these patterns must be replaced by one of the tests Karel can perform on its environment, as listed in Figure 1-4. Like function calls, tests include an empty set of parentheses, which are part of the Karel syntax. Every test in the list is paired with a second test that checks the opposite condition. For example, you can use the `frontIsClear` condition to check whether the path ahead of Karel is clear or the `frontIsBlocked` condition to see if there is a wall blocking the way. Choosing the right condition requires you to think about the logic of the problem and see which condition is easiest to test.

---

**Figure 1-4 Conditions that Karel can test**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>frontIsClear()</code></td>
<td>Is there a wall in front of Karel?</td>
</tr>
<tr>
<td><code>leftIsClear()</code></td>
<td>Is there a wall to Karel’s left?</td>
</tr>
<tr>
<td><code>rightIsClear()</code></td>
<td>Is there a wall to Karel’s right?</td>
</tr>
<tr>
<td><code>beepersPresent()</code></td>
<td>Are there beepers on this corner?</td>
</tr>
<tr>
<td><code>beepersInBag()</code></td>
<td>Any there beepers in Karel’s bag?</td>
</tr>
<tr>
<td><code>facingNorth()</code></td>
<td>Is Karel facing north?</td>
</tr>
<tr>
<td><code>facingEast()</code></td>
<td>Is Karel facing east?</td>
</tr>
<tr>
<td><code>facingSouth()</code></td>
<td>Is Karel facing south?</td>
</tr>
<tr>
<td><code>facingWest()</code></td>
<td>Is Karel facing west?</td>
</tr>
</tbody>
</table>
You can use the if statement to modify the definition of the `fillPothole` function so that Karel puts down a beeper only if there is not already a beeper on that corner. The new definition of `fillPothole` looks like this:

```java
function fillPothole() {
    turnRight();
    move();
    if (noBeepersPresent()) {
        putBeeper();
    }
    turnAround();
    move();
    turnRight();
}
```

The if statement in this example illustrates several features common to all control statements in Karel. The control statement begins with a header, which indicates the type of control statement along with any additional information to control the program flow. In this case, the header is

```java
if (noBeepersPresent())
```

which shows that the statements enclosed within the braces should be executed only if the `noBeepersPresent` test is true. The statements enclosed in braces represent the body of the control statement.

It often makes sense to include if statements in a function that check whether it makes sense to apply that function in the current state of the world. For example, calling the `fillPothole` function makes sense only if Karel is facing east directly above a hole. You can use the `rightIsClear` test to determine if there is a hole to the south, which is the direction to the right of the one that Karel is facing. The following implementation of `fillPothole` includes this test along with the `noBeepersPresent` test you have already seen:

```java
function fillPothole() {
    if (rightIsClear()) {
        turnRight();
        move();
        if (noBeepersPresent()) {
            putBeeper();
        }
        turnAround();
        move();
        turnRight();
    }
}
```
As you can see from the spacing used in this example, the body of each control statement is indented with respect to the statements that enclose it. The indentation makes it much easier to see exactly which statements will be affected by the control statement. Such indentation is particularly important when the body of a control statement contains other control statements. Control statements that occur inside other control statements are said to be **nested**.

**Iterative statements**

In solving Karel problems, you will often find that repetition is a necessary part of your solution. If you were really going to program a robot to fill potholes, it would hardly be worthwhile to have it fill just one. The value of having a robot perform such a task comes from the fact that the robot could repeatedly execute its program to fill one pothole after another.

To see how repetition can be used in the context of a programming problem, consider the following stylized roadway in which the potholes are evenly spaced along 1st Street at every even-numbered avenue:

```
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
```

Your mission is to write a program that instructs Karel to fill all the holes in this road. Note that the road reaches a dead end after 11th Avenue, which means that you have exactly five holes to fill.

Since you know from this example that there are exactly five holes to fill, the control statement that you need is a **repeat** statement, which specifies that you want to repeat some operation a predetermined number of times. The **repeat** statement looks like this:

```
repeat (number of repetitions) {
  statements to be repeated
}
```

For example, if you want to change the **fillTwoPotholes.k** program so that it solves the more complex problem of filling five evenly-spaced holes, all you have to do is write the following code:
function fillFivePotholes() {
    repeat (5) {
        move();
        fillPothole();
        move();
    }
}

The **repeat** statement is useful only when you know in advance the number of repetitions you need to perform. In most applications, the number of repetitions is controlled by the specific nature of the problem. For example, it seems unlikely that a pothole-filling robot could always count on there being exactly five potholes. It would be much better if Karel could continue to fill holes until it encountered some condition that caused it to stop, such as reaching the end of the street. Such a program would be more general in its application and would work correctly in either of the following worlds as well as any other world in which the potholes were spaced exactly two corners apart:

To write a general program that works with any of these worlds, you need to use a **while** statement. In Karel, a **while** statement has the following general form:

```karel
while (conditional test) {
    statements to be repeated
}
```

The conditional test in the header is chosen from the set of conditions listed in Figure 1-4.

To solve the pothole-filling problem, Karel needs to check whether the path in front is clear by invoking the condition `frontIsClear`. If you use the `frontIsClear` condition in a **while** loop, Karel will repeatedly execute the loop until it hits a wall. The **while** statement therefore makes it possible to solve the somewhat more general problem of repairing a roadway, as long as the potholes appear at every even-numbered corner and the end of the roadway is marked by a wall. The following definition of the function `fillRegularPotholes` accomplishes this task:
function fillRegularPotholes() {
    while (frontIsClear()) {
        move();
        fillPothole();
        move();
    }
}

Solving general problems

So far, the various pothole-filling programs have not been very realistic, because they rely on specific conditions—such as evenly spaced potholes—that are unlikely to be true in the real world. If you want to write a more general program to fill potholes, it should be able to work with fewer constraints. In particular, it does not really make sense to assume that the potholes occur on every other corner. Ideally, there should be no limits on the number of potholes or any restrictions on their spacing. A pothole is simply an opening in the wall representing the road surface.

To change the program so that it solves this more general problem requires you to think about the overall strategy in a different way. Instead of having a loop that cycles through each pothole, you need to have it call `fillPothole` at every intersection along the roadway.

This strategic analysis suggests that the solution to the general problem might be as simple as the following definition:

```java
function fillAllPotholes() {
    while (frontIsClear()) {
        fillPothole();
        move();
    }
}
```

Unfortunately, the solution is not quite so easy. The program as written contains a logical flaw—the sort of error that programmers call a **bug**. This book uses the bug symbol on the right to mark functions that contain errors to ensure that you don’t accidentally use those examples as models for your own code.

The bug in this example turns out to be relatively subtle. It would be easy to miss, even if you thought you had tested the program thoroughly. In particular, the program works correctly on all the pothole-filling worlds you’ve seen so far and on many which you haven’t. It only fails if there is a pothole in the very last avenue on the street, as illustrated by the following before-and-after diagram:
In this example, Karel stops without filling the last pothole. In fact, if you watch the execution carefully, Karel never even goes down into that last pothole to check whether it needs filling. What’s the problem here?

If you follow through the logic of the program carefully, you’ll discover that the bug lies in the structure of the loop in `fillAllPotholes`, which looks like this:

```java
while (frontIsClear()) {
    fillPothole();
    move();
}
```

As soon as Karel finishes filling the pothole on 6th Avenue, it executes the `move` instruction and returns to the top of the `while` loop. At that point, Karel is standing at the corner of 7th Avenue and 2nd street, where it is up against the boundary wall. Because the `frontIsClear` test now fails, the `while` loop exits without checking the last segment of the roadway.

The bug in this program is an example of a programming problem called a **fencepost error**. The name comes from the fact that it takes one more fence post than you might think to fence off a particular distance. How many fence posts, for example, do you need to build a 100-foot fence if the posts are always positioned 10 feet apart? The answer is 11, as illustrated by the following diagram:

![100 feet, 11 fenceposts](image)

The situation in Karel’s world has much the same structure. In order to fill potholes in a street that is seven corners long, Karel has to check for seven potholes but only has to move six times. Because Karel starts and finishes at an end of the roadway, it needs to execute one fewer `move` instruction than the number of corners it checks.

Once you discover it, fixing this bug is actually quite easy. Before Karel stops at the end of the roadway, all that the program has to do is to make a special-case check for a pothole at the final intersection, as follows:
function fillAllPotholes() {
    while (frontIsClear()) {
        fillPothole();
        move();
    }
    fillPothole();
}

The complete program appears in Figure 1-5.

```karel
/*
 * File: FillAllPotholes.k
 * ______________
 * This program fills an arbitrary number of potholes in a road.
 */
import "turns";

/*
 * Fills all the potholes up to the end of the road.
 */
function fillAllPotholes() {
    while (frontIsClear()) {
        fillPothole();
        move();
    }
    fillPothole();
}

/*
 * Fills a pothole immediately underneath Karel, if one exists.
 * When you call this function, Karel must be standing just above
 * the pothole, facing east. When the function returns, Karel
 * will be in its original position above the repaired pothole.
 */
function fillPothole() {
    if (rightIsClear()) {
        turnRight();
        move();
        if (noBeepersPresent()) {
            putBeeper();
        }
        turnAround();
        move();
        turnRight();
    }
}
```
1.4 Stepwise refinement

When you are faced with a complex programming problem, figuring out how to decompose the problem into pieces is usually one of your most important tasks. One of the most productive strategies is called stepwise refinement, which consists of solving problems by starting with the problem as a whole. You break the whole problem down into pieces, and then solve each piece, breaking those down further if necessary.

An exercise in stepwise refinement

Suppose that Karel is initially facing east at the corner of 1st Street and 1st Avenue in a world in which each avenue may contain a vertical tower of beepers of an unknown height, although some avenues may also be empty. Karel’s job is to collect the beepers in each of these towers, put them all back down on the easternmost corner of 1st Street, and then return to its starting position. Figure 1-6 illustrates the operation of this program for one possible world.

The key to solving this problem is to decompose the program in the right way. This task is more complex than the others you have seen, which makes choosing appropriate subproblems more important to obtaining a successful solution.

The principle of top-down design

The central idea in stepwise refinement is that you should start the design of your program from the top, which refers to the level of the program that is conceptually highest and most abstract. At this level, the beeper tower problem is clearly divided into three independent phases. First, Karel has to collect all the beepers. Second,
Karel has to deposit them on the last intersection. Third, Karel has to return to its home position. This outline suggests the following decomposition of the problem:

```
function collectBeeperTowers() {
    collectAllBeepers();
    dropAllBeepers();
    returnHome();
}
```

At this level, the problem is easy to understand. Even though you have not written the code for the functions in the body of `collectBeeperTowers`, it is important to convince yourself that, as long as you believe that the functions you are about to write will solve the subproblems correctly, you will have a solution to the problem as a whole.

**Refining the first subproblem**

Now that you have defined the structure for the program as a whole, it is time to move on to the first subproblem, which consists of collecting all the beepers. This task is itself more complicated than the problems you have seen so far. Collecting all the beepers means that you have to pick up the beepers in every tower until you get to the final corner. The fact that you need to repeat an operation for each tower suggests that you need to use a `while` loop.

But what does this `while` loop look like? First of all, you should think about the conditional test. You want Karel to stop when it hits the wall at the end of the row, which means that you want Karel to keep going as long as the space in front is clear. The `collectAllBeepers` function will therefore include a `while` loop that uses the `frontIsClear` test. At each position, you want Karel to collect all the beepers in the tower beginning on that corner. If you give that operation a name like `collectOneTower`, you can then write a definition for the `collectAllBeepers` function even though you haven’t yet filled in the details. You do, however, have to be careful. To avoid the fencepost problem described on page 17, the code must call `collectOneTower` after the last cycle of the loop, as follows:

```
function collectAllBeepers {
    while (frontIsClear()) {
        collectOneTower();
        move();
    }
    collectOneTower();
}
```

As you can see, this function has the same structure as the `fillAllPotholes` function in Figure 1-5. The only difference is that `collectAllBeepers` calls
collectOneTower where the earlier one called fillPothole. These two programs are each examples of a general strategy that looks like this:

```java
while (frontIsClear()) {
    Perform some operation.
    move();
}
```

Perform the same operation for the final corner.

You can use this strategy whenever you need to perform an operation on every corner as you move along a path that ends at a wall. If you remember the general strategy, you can quickly write the code whenever you encounter a problem of a similar form. Reusable strategies of this sort come up frequently in programming and are referred to as **programming idioms** or patterns. The more patterns you know, the easier it will be for you to find one that fits a particular type of problem.

### Coding the next level

Even though the code for collectAllBeepers is complete, you can’t run the program until you implement collectOneTower. When collectOneTower is called, Karel is standing either at the base of a tower or on an empty corner. In the former case, you need to collect the beepers in the tower. In the latter case, you can simply move on. This situation at first suggests that you need an if statement in which you call beepersPresent to see whether a tower exists.

Before you add such a statement to the code, it is worth giving some thought to whether you need to make this test. Often, programs can be made much simpler by observing that cases that at first seem to be special can be treated in precisely the same way as the more general situation. In the current problem, what happens if you decide that there is a tower of beepers on every avenue but that some of those towers are zero beepers high? Making use of this insight simplifies the program because you no longer have to test whether there is a tower on a particular avenue.

The collectOneTower function is still complex enough that an additional level of decomposition makes sense. To collect all the beepers in a tower, Karel has to climb the tower to collect each beeper, turn around, and then return to the wall that marks the southern boundary of the world. These steps suggest the following code:

```java
function collectOneTower() {
    turnLeft();
    collectLineOfBeepers();
    turnAround();
    moveToWall();
    turnLeft();
    turnLeft();
}
```
The `turnLeft` instructions at the beginning and end of the `collectOneTower` function are critical to the correctness of this program. When `collectOneTower` is called, Karel is always somewhere on 1st Street facing east. When it completes its operation, the program works correctly only if Karel is once again facing east. Conditions that must be true before a function is called are called *preconditions*; conditions that must apply after the function finishes are called *postconditions*.

**Finishing up**

Although the hard work has been done, a few loose ends still need to be resolved. Four functions—`collectLineOfBeepers`, `dropAllBeepers`, `moveToWall`, and `returnHome`—are as yet unwritten. Fortunately, each of these functions is simple enough to code without any further decomposition. A complete implementation of the `CollectBeeperTowers` program appears in Figure 1-7.

### 1.5 Algorithms in Karel’s world

Although top-down design is a critical strategy for programming, you can’t apply it mechanically without thinking about problem-solving strategies. Figuring out how to solve a particular problem generally requires considerable creativity. The process of designing a solution strategy is traditionally called *algorithmic design*.

---

**Figure 1-7** Karel program to collect all the beepers in a set of towers

```/*
 * File: CollectBeeperTowers.k
 * --------------------------
 * This program collects all the beepers in a series of towers, deposits
 * them at the easternmost corner on 1st Street, and then returns home.
 */

function collectBeeperTowers() {
    collectAllBeepers();
    dropAllBeepers();
    returnHome();
}

/*
 * Collects the beepers from every tower along 1st Street.
 */

function collectAllBeepers() {
    while (frontIsClear()) {
        collectOneTower();
        move();
    }
    collectOneTower();
}  ```
A Gentle Introduction

Figure 1-7 Karel program to collect all the beepers in a set of towers (continued)

    /*
     * Collects the beepers in a single tower.
     * /
     
    function collectOneTower() {
        turnLeft();
        collectLineOfBeepers();
        turnAround();
        moveToWall();
        turnLeft();
    }

    /*
     * Collects a consecutive line of beepers.
     * /
     
    function collectLineOfBeepers() {
        while (beepersPresent()) {
            pickBeeper();
            if (frontIsClear()) {
                move();
            }
        }
    }

    /*
     * Drops all the beepers from Karel's bag on the current corner.
     * /
     
    function dropAllBeepers() {
        while (beepersInBag()) {
            putBeeper();
        }
    }

    /*
     * Returns Karel to the corner of 1st Avenue and 1st Street, facing east.
     * /
     
    function returnHome() {
        turnAround();
        moveToWall();
        turnAround();
    }

    /*
     * Moves Karel forward until it is blocked by a wall.
     * /
     
    function moveToWall() {
        while (frontIsClear()) {
            move();
        }
    }
The word *algorithm* comes from the name of a ninth-century Persian mathematician, Muḥammad ibn Mūsā al-Khwārizmī, who developed the first systematic treatment of algebra. You will have more of a chance to learn about algorithms and al-Khwārizmī in Chapter 4.

Even before you have a chance to study algorithms in more detail, it is useful to consider a simple algorithm in Karel’s domain. Suppose, for example, that you want to teach Karel to escape from a maze. In Karel’s world, a maze might look like this:

Karel’s job is to navigate the corridors of the maze until it finds the beeper marking the exit. The program, however, must be general enough to solve any maze, and not just the one pictured here.

For most mazes, you can use a simple strategy called the *right-hand rule*, which you start by putting your right hand on the wall and then go through the maze without ever taking your hand off the wall. Another way to express this strategy is to proceed through the maze one step at a time, always taking the rightmost available path. The program that implements the right-hand rule turns out to be easy to implement in Karel and fits in a single function:

```karel
function solveMazeUsingRightHandRule() {
  while (noBeepersPresent()) {
    turnRight();
    while (frontIsBlocked()) {
      turnLeft();
    }
    move();
  }
}
```

At the beginning of the outer *while* loop, Karel turns right to check whether that path is available. The inner *while* loop then turns left until an opening appears.
When that happens, Karel moves forward, and the entire process continues until Karel reaches the beeper marking the end of the maze.

**Summary**

In this chapter, you had a chance to meet Karel, a very simple robot living in a very simple world. Starting off with Karel makes it possible to learn the fundamentals of programming without having to master the many complexities that come with a full-scale programming language. The important points in this chapter include the following:

- Karel the Robot is a *programming microworld* developed in the 1970s by Rich Pattis who was then a computer science graduate student at Stanford. Ever since that time, Karel has welcomed each new generation of Stanford students to the wonders of programming.
- Karel lives in a rectangular world defined by *streets* running from west to east and *avenues* running from south to north. Karel is always positioned on a *corner* marking the intersection of a street and an avenue and must be facing in one of the four standard compass directions (north, east, south, and west).
- Karel’s world is surrounded by *walls* around the border and may also contain additional interior walls that block Karel’s passage between two corners.
- Karel’s world can also contain *beepers*, which Rich Pattis describes as “plastic cones which emit a quiet beeping noise.” Beepers exist either on corners or in Karel’s beeper bag, both of which can contain an arbitrarily large number of beepers.
- When Karel is shipped from the factory, it knows how to execute only four operations—`move`, `turnLeft`, `putBeeper`, and `pickBeeper`—which are defined in detail in Figure 1-1 on page 4.
- You can extend Karel’s repertoire of operations by defining *functions*, which are sequences of operations that have been collected together and given a name. For example, the following function definition gives Karel the power to turn right by executing three consecutive left turns:

```plaintext
function turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
}
```

- The functions `turnRight` and `turnAround` are included in a *library* called *turns*, which you can *import* by including the following line in your program:

```plaintext
import "turns";
```
• The best strategy for solving a large problem is to divide it into successively smaller subproblems, each of which is implemented as a separate function. This process is called decomposition or stepwise refinement.

• The Karel programming language includes control statements that fall into two classes. Conditional statements allow you to execute other statements only if a particular condition holds. Iterative statements allow you to repeat a sequence of statements, either a specified number of times or as long as a condition holds.

• The rules for Karel’s control statements appear in the syntax boxes to the right.

• The conditions that Karel can test appear in Figure 1-4 on page 11.

• When you are using iterative statements, it is important to avoid the fencepost error, which occurs when you fail to recognize that the number of move instructions necessary to cover a distance is one less than the number of corners.

• In computer science, an algorithm is a solution strategy. If you study computer science, algorithms will be one of the most important topics.

### Review questions

1. In your own words, explain the meaning and purpose of a programming microworld.

2. Who created the Karel microworld?

3. What is the etymology of the name Karel?

4. Define each of the following aspects of Karel’s world: street, avenue, corner, wall, and beeper.

5. What are the four predefined Karel functions?

6. What are the two functions included in the Karel library named turns?

7. What is the meant by the strategy of stepwise refinement?

8. What statement does Karel offer to execute statements only if some condition applies? What are the two forms of this statement?

9. What two statements does Karel offer for repeating a group of statements?

10. What condition would you use to test whether Karel can move forward from its current position? What condition would you use to test whether there are any beepers on the current corner?

11. What is a fencepost error?
12. What are preconditions and postconditions?

13. The `collectLineOfBeepers` function in Figure 1-7 includes an if statement that checks the `frontIsClear` condition before moving. Why is it important to make this test?

### Exercises

1. Only one of the two functions in the `turns` library is defined explicitly in this chapter. Write a Karel function that implements the other.

2. Suppose that Karel has settled into its house, which is the square area in the center of the following diagram:

```
+ + + + + + + +
+ + + + + + + +
+ + + + + + + +
+ + + + + + + +
+ + + + + + + +
```

Karel starts off in the northwest corner of its house as shown in the diagram. The problem is to program Karel to collect the newspaper—represented (as all objects in Karel’s world are) by a beeper—from outside the doorway and then to return to its initial position.

This exercise is extremely simple and is intended mostly to get you started. You can assume that every part of the world looks just as it does in the diagram. The house is exactly this size, the door is always in the position shown, and the beeper is just outside the door. Thus, all you have to do is write the sequence of statements necessary to have Karel perform the following tasks:

1. Move to the newspaper.
2. Pick it up.
3. Return to its original starting point.

Even though the program requires just a few lines, it is still worth getting at least a little practice in decomposition. In your solution, decompose the program so that it includes a function for each step shown in the outline.
3. Write a program that teaches Karel to climb a mountain exactly like this:

The steps involved are
1. Move up to the mountain.
2. Climb each of the four stair steps to reach the summit.
3. Plant a flag (represented by a beeper, of course) at the top of the mountain.
4. Climb down each of the four stair steps on the opposite side.
5. Move forward to the east end of the world.

The final state of the world should look like this:

4. Generalize the program you wrote in exercise 3 so that Karel is able to climb a stair-step mountain of any height. Thus, in addition to climbing the mountain in that exercise, it should be able to scale a molehill like
or an Everest-sized peak like

![Diagram of a tree with a cluster of four leaves arranged in a square]

5.

"sweet spring is your
time is my time is our
time for springtime is lovetime
and viva sweet love"

—e. e. cummings

For those who live in colder climates, winter can be a bitter time. The trees have lost their leaves and stand as empty monuments to the ravages of the season, as shown in the following sample world:

![Diagram of a tree with a cluster of four leaves arranged in a square]

In this sample world, the vertical wall sections represent barren tree trunks. Karel’s job is to climb each of the trees and adorn the top of each tree with a cluster of four leaves arranged in a square like this:
Thus, when Karel is done, the scene will look like this:

```
+ + + +   + + + +   + + + +   + + + +
+ + + +   + + + +   + + + +   + + + +
+ + + +   + + + +   + + + +   + + + +
+ + + +   + + + +   + + + +   + + + +
+ + + +   + + + +   + + + +   + + + +
+ + + +   + + + +   + + + +   + + + +
+ + + +   + + + +   + + + +   + + + +
```

The situation that Karel faces need not match exactly the one shown in the diagram. There may be more trees; Karel simply continues the process until there are no beepers left in the beeper bag. The trees may also be of different heights or spaced differently than the ones shown in the diagram. Your task is to design a program that is general enough to solve any such problem, subject to the following assumptions:

- Karel starts at the origin facing east, somewhere west of the first tree.
- The trees are always separated by at least two corners, so that the leaves at the top don’t interfere with one another.
- The trees always end at least two corners below the top, so that the leaf cluster will not run into the top wall.
- Karel has just enough beepers to outfit all the trees. The original number of beepers must therefore be four times the number of trees.
- Karel should finish facing east at the bottom of the last tree.

Think hard about what the parts of this program are and how you could break it down into simpler subproblems. What if there were only one tree? How does that simplify the problem, and how can you use the one-tree solution to help solve the more general case?

6. In this problem, your job is to program Karel to create a calendar by putting down beepers in a pattern that corresponds to the days in a particular month. Karel begins the task in the upper left row of a 7×6 world. On one of the intersections on that row—corresponding to the first day of the month in
A Gentle Introduction

question—there is a beeper pile containing exactly the same number of beepers as there are days in the month. For example, for a February that begins on a Monday and has 28 days, the initial state of the world would look like this:

```
1 2 3 4 5 6 7
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
```

What Karel needs to do is

1. Walk across the top row to find the beeper pile.
2. Pick up all the beepers.
3. Put the beepers down, one at a time, starting at the intersection on which it found the pile and then continuing across each row in turn until it runs out of beepers.

Thus, given the starting configuration for February, Karel should finish with a world diagram that looks like this:

```
1 2 3 4 5 6 7
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
+ + + + + + +
```
Karel may count on the following facts about the world:

- Karel’s world always has seven avenues and six streets.
- Karel begins at the corner of 6th Street and 1st Avenue, facing east, with an empty beeper bag.
- There is a pile with the correct number of beepers somewhere on 6th Street.
- At the end of execution, Karel should be positioned on top of the beeper representing the last day of the month, facing east.

7. More than a decade after Hurricane Katrina, considerable damage remains along the Gulf Coast, and some communities have yet to be rebuilt. As part of its plans to improve the nation’s infrastructure, the government has established a new program named Katrina Automated RELief (or KAREL) whose mission is to dispatch house-building robots to repair the damaged area. Your job is to program those robots.

Each robot begins at the west end of a street that might look like this:

```

```

Each beeper in the figure represents a pile of debris where a house once stood. Karel’s job is to walk along the street and build a new house in the place marked by each beeper. Those houses, moreover, need to be raised on stilts to avoid damage from the next storm. Each house, in fact, should look exactly like this:

```

```

The new house should be centered at the point at which the bit of debris was left, which means that the first house in the diagram above will be constructed with its left edge along 2nd Avenue.
At the end of the run, Karel should be at the east end of the street having created a set of houses that look like this for the initial conditions shown:

In solving this problem, you can count on the following facts about the world:

- Karel starts off facing east at the corner of 1st Street and 1st Avenue with an infinite number of beepers in its beeper bag.
- The beepers indicating the positions at which houses should be built will be spaced so that there is room to build the houses without overlapping or hitting walls.
- Karel must end up facing east at the southeast corner of the world. Moreover, Karel should not run into a wall if it builds a house that extends into that final corner.

8. Suppose that it’s Halloween, and Karel is going Trick-or-Treating. Karel starts off at the west end of a dead-end street that contains houses on both sides of street, such as the one pictured in the following diagram:

Each house has a front porch at some point along its front side. Karel’s mission is to go to each house, step into the porch area, and see if the porch contains a treat, represented by a beeper. If there is, Karel should pick it up. If not, Karel
should move on to the next house. Karel must check every porch on both sides of the street and should end up at the original intersection facing in the opposite direction. Thus, after executing your program in the world shown above, Karel should end up in the following position:

```
Karel may count on the following facts about the world:

• Karel starts out at the west end of a street, facing east, with an empty beeper bag. You do not know how long the street is, but you do know that there are walls closing off each end of the street.

• The houses on the street are packed closely together, with no space between adjacent houses.

• There may be any number of houses on the street. Moreover, the number of houses on one side is not necessarily equal to the number of houses on the other side. The individual houses typically vary in size.

• The side of the house facing the street is a solid wall except for a small porch, which is always one intersection wide. The porch can appear at any point in the front wall.

• Each porch is either empty or marked with a single beeper, representing a Halloween treat.

• At the end of execution, Karel should return to its original intersection at the west end of the street, but should now be facing west.
```

9. Now that Karel has mastered Halloween, it’s time to celebrate a different holiday. Karel has decided to deliver beeper valentines to every student in an elementary school class that is using Karel to learn about programming. Karel does not remember exactly how many desks there are in each horizontal row but does remember that there are precisely three rows of desks and that the classroom looks something like the one shown in the following diagram:
Karel may count on the following facts:

- Karel starts at 1\textsuperscript{st} Avenue and 1\textsuperscript{st} Street, facing east, with an infinite number of beepers in its bag.

- There are exactly three rows of student desks, positioned as shown in the diagram, just to the south of 3\textsuperscript{rd}, 5\textsuperscript{th}, and 7\textsuperscript{th} Streets.

- Karel does not know how many desks there are in each row (which may not all be the same), or how many blank spaces there are between the desks, or how many spaces exist between the desks at the ends of each row and the walls of the classroom. What Karel does know is that each of the desks is exactly one unit wide and that there are no desks right up against the wall.

When Karel is done, all of the desks in the room should have a valentine, as shown in the following diagram:
10. Having heard that programming is at least as much an art as a science, Karel has decided to enroll in a paint-by-numbers class. In this class, Karel is presented with a “canvas” containing piles of beepers, such as those shown in the following diagram:

To complete the paint-by-numbers task, all Karel has to do is walk from left to right across each street, pick up each pile of beepers, and then redistribute the beepers from that pile, one at a time, on each successive corner.

To get a sense of how this process works, consider what happens when Karel gets to 11th Street. At the beginning of the row, Karel is standing on the first corner with an empty beeper bag, as follows:

Karel begins by walking down the street looking for a beeper pile. The first one it finds is the pile of three beepers on 4th Avenue. When Karel gets to that corner, it picks up the beepers. This step leaves Karel in the following position with three beepers in its bag:
From here, the next step is to put the beepers down, one at a time, starting with the corner in which the pile was found. Executing this step leads to the following configuration, where Karel again has an empty beeper bag:

```
11 + + + □□□□ □ + + + + + +
```

Karel then repeats this process for the second beeper pile, ending up in the following position at the end of the row:

```
11 + + + □□□□ + □□□□ + + □
```

The final state of the world should look like this:

```
12 + + + + + + + + + + + + +
11 + + + □□□□ + □□□□ + + +
10 + + □□□□ □□□□ + □□□□ + +
 9 + □□□□ □□□□ □□□□ + □□□□ +
 8 + □□□□ □□□□ □□□□ □□□□ + +
 7 + □ □□□ □□□□ □□□□ □□□□ + +
 6 + □□□□ □□□□ □□□□ □□□□ □□□□ +
 5 + □ □□□ □□□□ □□□□ □□□□ □□□□ +
 4 + □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ +
 3 + □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ +
 2 + □ □□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ +
 1 □ □□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ +
```

Your job is to write the program that converts a paint-by-number picture of this sort into the corresponding completed masterpiece. In writing the program, Karel can count on the following facts about the world:

- The world contains an arbitrary number of beeper piles but no interior walls.
- The beeper piles never have so many beepers that they cause Karel to run into a wall or another beeper pile.
• Karel always starts facing east in the southwest corner (1st Street and 1st Avenue) with an empty beeper bag.

• Karel must finish execution facing east at the northeast corner of the world.

11. In this exercise, your job is to get Karel to create a checkerboard pattern of beepers inside an empty rectangular world, as illustrated in the before-and-after diagram in Figure 1-8.

This problem has a nice decomposition structure along with some interesting algorithmic issues. As you think about how you will solve the problem, you should make sure that your solution works with checkerboards that are different in size from the standard 8×8 checkerboard shown in the example. Odd-sized checkerboards are tricky, and you should make sure that your program generates the following pattern in a 5×3 world:

Another special case you need to consider is that of a world which is only one column wide or one row high.
12. Program Karel to place a single beeper at the center of 1st Street. For example, if Karel starts in the world

```
+ + + + +
+ + + + +
+ + + + +
+ + + + +
+ + + + +
```

it should end with Karel standing on a beeper in the following position:

```
+ + + + +
+ + + + +
+ + + + +
+ + + + +
+ + + + +
```

Note that the final configuration of the world should have only a single beeper at the midpoint of 1st Street. Along the way, Karel is allowed to place additional beepers wherever it wants to, but must pick them all up again before it finishes.

In solving this problem, you may count on the following:

- Karel starts at 1st Avenue and 1st Street, facing east, with an infinite number of beepers in its bag.
- The initial state of the world includes no interior walls or beepers.
- The world need not be square, but you may assume that it is at least as tall as it is wide.
- If the width of the world is odd, Karel must put the beeper in the center square. If the width is even, Karel may drop the beeper on either of the two center squares.
- It does not matter which direction Karel is facing at the end of the run.

There are many different algorithms you can use to solve this problem. The interesting part of this problem is to come up with a strategy that works.
CHAPTER 2

Introducing JavaScript

Computer programs are the most complex things that humans make.


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:

\[0\quad 42\quad -273\quad 3.14159265\quad -0.5\quad 1000000\]

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.

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:

\[
\begin{array}{c}
342 \\
- 173 \\
\hline
169
\end{array}
\]

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

```
> 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)
- $\ast$ 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:

```
> (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:

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

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
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:

```
name
value
```

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:

```
| r | 10 |
```

**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

```
let r = 2.5;
```

the value in the box would change as follows:

```
| r | 2.5 |
```

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
Introducing JavaScript

\[ balance += deposit; \]

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.

<table>
<thead>
<tr>
<th>Figure 2-2</th>
<th>Reserved words in JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>default</td>
</tr>
<tr>
<td>arguments</td>
<td>delete</td>
</tr>
<tr>
<td>await</td>
<td>do</td>
</tr>
<tr>
<td>boolean</td>
<td>double</td>
</tr>
<tr>
<td>break</td>
<td>else</td>
</tr>
<tr>
<td>byte</td>
<td>enum</td>
</tr>
<tr>
<td>case</td>
<td>eval</td>
</tr>
<tr>
<td>catch</td>
<td>export</td>
</tr>
<tr>
<td>char</td>
<td>extends</td>
</tr>
<tr>
<td>class</td>
<td>false</td>
</tr>
<tr>
<td>const</td>
<td>final</td>
</tr>
<tr>
<td>continue</td>
<td>finally</td>
</tr>
<tr>
<td>debugger</td>
<td>float</td>
</tr>
</tbody>
</table>


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 $\pi$. 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:

```
> 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:

```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:

```
> 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 \( \text{PI} \) 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:

```javascript
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:

<table>
<thead>
<tr>
<th>JavaScript Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; circleArea(1)</td>
</tr>
<tr>
<td>3.14159265</td>
</tr>
<tr>
<td>&gt; circleArea(10)</td>
</tr>
<tr>
<td>314.1592653</td>
</tr>
</tbody>
</table>

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:

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

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

<table>
<thead>
<tr>
<th>JavaScript Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; celsiusToFahrenheit(0)</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>&gt; celsiusToFahrenheit(20)</td>
</tr>
<tr>
<td>68</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
</tbody>
</table>
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

```javascript
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

<table>
<thead>
<tr>
<th>Mathematical constants</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.PI</td>
<td>The mathematical constant π.</td>
</tr>
<tr>
<td>Math.E</td>
<td>The mathematical constant e, which is the base for natural logarithms.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>General mathematical functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.abs(x)</td>
<td>Returns the absolute value of x.</td>
</tr>
<tr>
<td>Math.max(x, y, ...)</td>
<td>Returns the largest of the arguments.</td>
</tr>
<tr>
<td>Math.min(x, y, ...)</td>
<td>Returns the smallest of the arguments.</td>
</tr>
<tr>
<td>Math.sqrt(x)</td>
<td>Returns the square root of x.</td>
</tr>
<tr>
<td>Math.round(x)</td>
<td>Returns the closest integer to x.</td>
</tr>
<tr>
<td>Math.floor(x)</td>
<td>Returns the largest integer less than or equal to x.</td>
</tr>
<tr>
<td>Math.ceil(x)</td>
<td>Returns the smallest integer greater than or equal to x.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logarithmic and exponential functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.exp(x)</td>
<td>Returns the exponential function of x ($e^x$).</td>
</tr>
<tr>
<td>Math.log(x)</td>
<td>Returns the natural logarithm (base $e$) of x.</td>
</tr>
<tr>
<td>Math.log10(x)</td>
<td>Returns the common logarithm (base 10) of x.</td>
</tr>
<tr>
<td>Math.pow(x, y)</td>
<td>Returns $x^y$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trigonometric functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.cos(theta)</td>
<td>Returns the cosine of the radian angle $\theta$.</td>
</tr>
<tr>
<td>Math.sin(theta)</td>
<td>Returns the sine of the radian angle $\theta$.</td>
</tr>
<tr>
<td>Math.tan(theta)</td>
<td>Returns the tangent of the radian angle $\theta$.</td>
</tr>
<tr>
<td>Math.atan(x)</td>
<td>Returns the principal arctangent of $x$, which lies between $-\pi/2$ and $+\pi/2$.</td>
</tr>
<tr>
<td>Math.atan2(y, x)</td>
<td>Returns the angle between the x-axis and the line from the origin to $(x, y)$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Random number generator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.random()</td>
<td>Returns a random number that is at least 0 but strictly less than 1.</td>
</tr>
</tbody>
</table>
class, which is defined to be as close an approximation as possible to the mathematical constant $\pi$. 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 $(x, y)$:

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

## 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

```javascript
let name = "Eric";
```
Introducing JavaScript 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:

```
name
"Eric"
```

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:

```javascript
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:

```
ALPHABET
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

### 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

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

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

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

**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) $6.02252 \times 10^{23}$
   - 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 $\pi$. In the case of $\pi$, 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) $2 + 2 \ast (2 \ast 2 - 2) \% 2 / 2$
   - c) $10 + 9 \ast ((8 + 7) \% 6) + 5 \ast 4 \% 3 \ast 2 + 1$
   - d) $1 + 2 + (3 + 4) \ast ((5 \ast 6 \% 7 \ast 8) - 9) - 10$
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

1. How would you implement the following mathematical function in JavaScript:
   \[ f(x) = x^2 - 5x + 6 \]
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(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:

![Diagram of a triangle with base and height labeled](image)

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("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 negated by adding the affix \textit{un}-, or could be strengthened by the affix \textit{plus}-, or, for still greater emphasis, \textit{doubleplus}-. Thus, for example, \textit{uncold} meant “warm,” while \textit{pluscold} and \textit{doublepluscold} meant, respectively, “very cold” and “superlatively cold.”

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

```
<table>
<thead>
<tr>
<th>Newspeak</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; negate(&quot;cold&quot;)</td>
</tr>
<tr>
<td>uncold</td>
</tr>
<tr>
<td>&gt; intensify(&quot;cold&quot;)</td>
</tr>
<tr>
<td>pluscold</td>
</tr>
<tr>
<td>&gt; reinforce(intensify(&quot;cold&quot;))</td>
</tr>
<tr>
<td>doublepluscold</td>
</tr>
<tr>
<td>&gt; reinforce(intensify(negate(&quot;good&quot;)))</td>
</tr>
<tr>
<td>doubleplusungood</td>
</tr>
</tbody>
</table>
```

CHAPTER 3

Running Programs in the Browser

The Web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past.

— Tim Berners-Lee, 18th World Wide Web Conference, 2009

Sir Tim Berners-Lee (1955–)

Tim Berners-Lee graduated from Oxford University with a degree in physics and went on to become a research fellow at CERN, the international nuclear research lab near Geneva, Switzerland. In March 1989, Berners-Lee wrote a proposal for a new set of communication protocols that would allow users to navigate easily through a large collection of data repositories stored on many different computers. That vision became the World Wide Web, now used by billions of people throughout the world. Throughout the web’s history, Berners-Lee has campaigned to ensure that access to the web remains free and open, unrestricted by either government or corporate control. For his pioneering contributions, Berners-Lee was knighted by Queen Elizabeth II in 2004 and received the Turing Award, the computing field’s highest honor, in 2016.
Although the JavaScript console used in Chapter 2 allows you to see how JavaScript evaluates expressions and simple functions, it does not give you a sense of how JavaScript runs a complete program. As is usually the case when you are learning about programming, the best way to learn how JavaScript programs work is to look at an example. The JavaScript program in the following section provides a simple but nonetheless powerful foundation that you can easily modify to work with the other programs you will see in this text. At the same time, this example also introduces a bit of cultural history that all computer science students should see at some point.

### 3.1 The “Hello World” program

JavaScript is only one of a collection of many programming languages that traces its roots to C, one of the most successful programming languages in the history of computing. In the book that serves as C’s defining document, *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the authors offer the following advice on the first page of Chapter 1:

> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

```plaintext
Print the words hello, world
```

This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where the output went. With these mechanical details mastered, everything else is comparatively easy.

That advice was followed by the four-line text of the “Hello World” program, which became part of the heritage shared by all C programmers.

**The JavaScript implementation of “Hello World”**

JavaScript, of course, is different from C, and the “Hello World” program will not look exactly the same in the two languages. Even so, the underlying advice remains sound: the first program you write should be as simple as possible so that you can focus your attention on the mechanics of the programming process. Your mission—and you *should* definitely decide to accept it—is to get the JavaScript version of “Hello World” running. The JavaScript version of the program, complete with explanatory comments that acknowledge the debt to Kernighan and Ritchie, appears in Figure 3-1. Outside of the commentary, the program itself consists of one function definition whose body is one line long, as follows:

```javascript
function HelloWorld() {
    console.log("hello, world");
}
```
3.1 The “Hello World” program

The body of the `HelloWorld` function calls the built-in function `console.log` and asks it to display the string “hello world” on the JavaScript console.

So far, everything seems reasonably straightforward. As Kernighan and Ritchie suggest, however, the hard parts lie in figuring out how “to create the program text somewhere, compile it successfully, load it, run it, and find out where the output went.” Those operations—which would no longer involve exactly the same steps as in the time Kernighan and Ritchie were writing—differ depending on what programming tools you happen to be using.

You will need at least two applications to get started. First, you need a text editor, which will allow you to create JavaScript program files. All modern computers come with some kind of text editor, but you will find it easier to write your programs if the editor you use understands the structure of JavaScript well enough to catch simple typographical errors and help you understand the different programming constructs by displaying them in colors that indicate their function. Second, you need a web browser that can read and display web pages. It doesn’t matter which browser you use as long as it is modern enough to interpret JavaScript Version 6, which was released in 2015. If you are using a browser that is older than that, you should update your browser to the current version.

The first thing you need to do is use your editor to type in the `HelloWorld.js` program exactly as it appears in Figure 3-1 and then save it in a new folder on your computer. That step, however, only gets you part of the way toward running the program in the browser. To complete the task, you need to learn more about the structure of the web and how to embed JavaScript programs within a web page.

**JavaScript and the web**

Everyone who uses computers today is familiar with the World Wide Web, the vast constellation of interconnected documents accessible on the computer networks that

---

**FIGURE 3-1** The “Hello World” program in JavaScript

```javascript
/*
 * File: HelloWorld.js
 * -------------
 * This program displays "hello, world" on the console. It is inspired
 * by the first program in Brian Kernighan and Dennis Ritchie's classic
 */

function HelloWorld() {
    console.log("hello, world");
}
```
Running Programs in the Browser

span the globe. At some point in 2014, the number of web sites passed the one-billion mark and has continued to grow rapidly since then. Each page is identified by a **uniform resource locator** or **URL** that serves as its address. Most web pages contain embedded references to other pages on related topics. These references are called **hyperlinks** and give the web its interconnected structure.

When you enter an explicit URL into your browser or click on a hyperlink containing an embedded URL, the browser fetches the contents of the web page at that address. For most web pages, the browser uses an interaction scheme called the **Hypertext Transfer Protocol**—indicated by the `http:` prefix at the beginning of a typical URL—to read the contents of the page. The browser then interprets the content of the page and displays it on the screen.

Modern web pages use three distinct but interrelated technologies to define the contents of the page:

1. The structure and contents of the page are defined using a file written using the **Hypertext Markup Language** or **HTML**.
2. The visual appearance of the page is specified using **Cascading Style Sheets** or **CSS**.
3. Any interactive behavior of the page is represented using one or more files, which are conventionally written in **JavaScript**.

If you want to create professional-quality web pages, you will need to learn more about all three of these technologies. Because this book focuses on programming in **JavaScript**, it presents only enough about **HTML** and **CSS** to allow you to write simple web pages that run **JavaScript** programs.

**An HTML template for JavaScript programs**

Every web page is associated with an HTML file—which, by convention, is usually named `index.html`—that describes the contents of the page. The content of the `index.html` file consists of text that conforms to the syntactic rules of HTML. In particular, the `index.html` file is organized into a series of sections marked by keywords enclosed in angle brackets, which are called **tags** in HTML. As you will see in the examples later in this section, some tags include additional information before the closing angle bracket. These additional fields are called **attributes**.

The `index.html` file begins with a special tag that marks the file as a standard HTML index:

```
<!DOCTYPE html>
```
After the `<!DOCTYPE>` tag, HTML tags usually occur in pairs. The first tag opens a section of the HTML file. The second tag, which uses the same keyword preceded by a slash character, closes that section. For example, the entire HTML text in the `index.html` file begins with a tag named `<html>` and ends with the corresponding closing tag `</html>`. To make it clear to the reader exactly what parts of the HTML file are included within each pair of tags, the lines between the opening and closing tag are typically indented.

A standard HTML file includes two sections between the `<html>` and `</html>` markers. The first of these is the `<head>` section, which defines a few overarching features of the page; the second is the `<body>` section, which defines the page contents. As with other paired tags, the `<head>` and `<body>` sections end with the tags `</head>` and `</body>`, respectively.

For simple JavaScript-based web pages, the `<head>` section contains encloses two types of interior tags. The first of these is the `<title>` section, which defines the title that appears at the top of the web page. The `<title>` section has the form

```html
<title>whatever title you want to use</title>
```

where you can replace the italicized text with whatever text you want to use as the title. By convention, the web programs in this book use the name of the program file as the title, so that the `<title>` section for the `HelloWorld.js` program would be

```html
<title>HelloWorld</title>
```

The other components of the `<head>` section are one or more `<script>` tags that specify the names of JavaScript files to load. Each of these `<script>` tags has the following form:

```html
<script type="text/javascript" src="filename"></script>
```

In this pattern, you need to replace the italicized `filename` marker with the actual name of the file. To load the `HelloWorld.js` file, for example, you would need to specify the following `<script>` tag:

```html
<script type="text/javascript" src="HelloWorld.js"></script>
```

You also need to include `<script>` tags in the `<head>` section to load any JavaScript libraries your program requires. And while the simplest version of the “Hello World” program may not technically require any libraries, it turns out that adding a library to this section will make your life as a programmer much easier. Remember that one of your tasks in Kernighan and Ritchie’s checklist is to “find out where the output went.” Most browsers make the console log hard to find to
minimize confusion for the average web user, who could easily be distracted by messages appearing in the console log. To make console output easier to find, the index.html files used in this book include the following <script> tag to load a library called JSConsole.js, which displays the console log as part of the web page itself:

```html
<script type="text/javascript" src="JSConsole.js"></script>
```

For simple JavaScript-based web pages that contain no other content, the <body> section will be empty, with nothing between the opening and closing tags. The <body> tag, however, must specify an onload attribute to get the program started. The value of the onload attribute is a JavaScript expression, which is ordinarily a function call. For example, to trigger a call to the HelloWorld function when the page has finished loading all the necessary JavaScript code, the onload attribute would have the value "HelloWorld()".

The complete contents of the index.html file for the “Hello World” program appear in Figure 3-2. You can use this file as a template for the index.html files you need to implement other JavaScript-based web pages.

### 3.2 Introducing the graphics library

Although it is possible to learn the fundamentals of programming using only the numeric and string types you saw in Chapter 2, numbers and strings are not as exciting as they were in the early years of computing. For students who have grown up in the 21st century, much of the excitement surrounding computers comes from their ability to work with other more interesting types of data, including images and interactive graphical objects. JavaScript—particularly given that it has become the leading language for programming content on the web—is ideal for working with graphical data. Moreover, introducing just a few graphical types makes it possible to create applications that are much more engaging and give you a greater incentive to master the material.

**Figure 3-2** The index.html file for “Hello World”

```html
<!DOCTYPE html>
<html>
  <head>
    <title>HelloWorld</title>
    <script type="text/javascript" src="JSConsole.js"></script>
    <script type="text/javascript" src="HelloWorld.js"></script>
  </head>
  <body onload="HelloWorld()"></body>
</html>
```
This rest of this chapter introduces a subset of the facilities available in the Stanford Graphics Library, which is a collection of graphical tools that allow you to create simple graphical applications. The discussion in this chapter is intended to provide enough information to get you started. The rest of the graphics library is introduced in later chapters as those features are needed.

A more modern version of “Hello World”

In much the same way that HelloWorld.js was a useful program to illustrate the use of JavaScript with a web-based console, it makes sense to use the same problem as a starting point for graphical programs in JavaScript. The new goal is no longer to print the words “hello, world” but instead to display those words in a graphics window embedded in the web page. The code for the GraphicsHelloWorld.js program needed to accomplish this task appears in Figure 3-3.

Like the HelloWorld.js program in Figure 3-1, GraphicsHelloWorld.js is designed to run in the browser and therefore needs to have an index.html file that defines the structure of the web page. The file is almost exactly the same as the one for HelloWorld.js. The only difference is that this program needs to load the graphics library instead of the console library. The corresponding <script> tag looks like this:

```
<script type="text/javascript" src="JSGraphics.js"></script>
```

The complete index.html file appears in Figure 3-4 at the top of the next page.

---

**Figure 3-3** A graphical version of the “Hello World” program

```javascript
/*
 * File: GraphicsHelloWorld.js
 * _______________________
 * This program displays the string "hello, world" at location (50, 100)
 * on the graphics window. The inspiration for this program comes from
 */

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 200;

/* Main program */

function GraphicsHelloWorld() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let msg = GLabel("hello, world", 50, 100);
    gw.add(msg);
}
```
Running Programs in the Browser

The main function for the `GraphicsHelloWorld.js` program looks like this:

```javascript
function GraphicsHelloWorld() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let msg = GLabel("hello, world", 50, 100);
    gw.add(msg);
}
```

The body of `GraphicsHelloWorld` begins with two variable declarations, one for the variable `gw`, which stands for “graphics window,” and one for the variable `msg`, which refers to the message on the screen. The declarations themselves have the same form as the ones you have seen earlier. Each declares a variable and initializes it to a value. What’s different is the type of these values.

**Classes, objects, and methods**

The values stored in the variables `gw` and `msg` are more complex than the numbers and strings you’ve worked with so far, but the underlying principles are the same. For example, the declaration

```javascript
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
```

creates a variable named `gw` and initializes it to a value that gives the programmer access to a graphics window created within the web page. The parameters—which are defined as constants to make them easier to change—indicate the size of the window, measured in units called pixels, which are the tiny dots that cover the face of the display. The values of these constants therefore create a `GWindow` that is 500 pixels wide and 200 pixels high.

In JavaScript, a value that represents some usually larger and more complex value is called a **reference**. In this case, the variable `gw` is initialized to contain a reference to a portion of the browser window capable of displaying graphical objects, as illustrated by the following diagram:
3.2 Introducing the graphics library

As the arrow suggests, the reference stored in \texttt{gw} points to a larger value that represents the graphics window on the screen. The data value representing the window is an example of what computer scientists call an \textit{object}, which is a conceptually integrated entity that ties together the information that defines the state of the object and the operations that affect that state. Each object in JavaScript is a representative of a \textit{class}, which is easiest to imagine as a template that defines the attributes and operations shared by all objects of a particular type. A single class can give rise to many different objects; each such object is said to be an \textit{instance} of that class.

The second line in the function

\begin{verbatim}
let msg = GLabel("hello, world", 50, 100);
\end{verbatim}

operates in a similar fashion. This line creates a \texttt{GLabel} object whose internal state includes the string to be displayed in the label and the coordinates at which the label should appear. This declaration creates a reference, which looks something like this:

In addition to the text of the message and the coordinate values, the \texttt{GLabel} object also contains the code necessary to make the message appear on the graphics window, even though you won’t actually see that code unless you look inside the graphics library. The internal data values and the associated code are not available to the function that creates the \texttt{GLabel} but are instead securely packaged inside the object. This model of packaging together data and code is called \textit{encapsulation}.

Even though the declarations of the variables \texttt{gw} and \texttt{msg} create the necessary objects, these lines alone do not cause the \texttt{GLabel} to appear in the \texttt{GWindow}. To get the message to appear, the program has to tell the \texttt{GWindow} object stored in \texttt{gw} to add the \texttt{GLabel} stored in \texttt{msg} to its internal list of graphical objects to display on the window. This step in the process is the responsibility of the last line in the \texttt{GraphicsHelloWorld.js} program, which looks like this:

\begin{verbatim}
gw.add(msg);
\end{verbatim}

Understanding how this statement works requires you to understand a little more about the way that JavaScript works with objects.
Sending messages to objects

When you are programming in a language that supports objects, it is useful to adopt at least some of the ideas and terminology of the object-oriented paradigm, a conceptual model of programming that focuses on objects and their interactions rather than on the more traditional model in which data and operations are seen as separate. In object-oriented programming, the generic term for anything that triggers a particular behavior in an object is called a message. In JavaScript, the object-oriented idea of sending a message to an object is implemented by calling a function associated with that object. Functions that are associated with an object are called methods, and the object on which the method is invoked is called the receiver. In JavaScript, method calls use the following syntax:

receiver.name(arguments)

In the method call `gw.add(msg)`, the graphics window stored in `gw` is the receiver, and `add` is the name of the method that responds to the message. The argument `msg` lets the implementation of the `GWindow` class know what graphical object to add, which in this case is the `GLabel` stored in the `msg` variable. The `GWindow` responds by displaying the message at the specified coordinates on the screen, which creates the following image:

As you can see from the screen image, the desired message is there. It’s not very large or exciting, but you’ll have a chance to spice it up a bit later in the chapter.

Creating objects

The `GraphicsHelloWorld.js` program includes two lines that create new objects:

```javascript
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let msg = GLabel("hello, world", 50, 100);
```

The functions `GWindow` and `GLabel` are part of the definition of the `GWindow` and `GLabel` classes in the `JSGraphics.js` library and serve to create new objects of the appropriate type. Functions that create new objects are called factory methods and typically start with an uppercase letter.
3.3 Classes in the graphics library

The `GLabel` class introduced in the preceding section is only one of several classes in the graphics library that represents an object you can display on the screen. This section introduces three other classes—`GRect`, `GOval`, and `GLine`—that, together with `GLabel` and `GWindow`, provide a wonderful “starter kit” for creating graphical applications. You will have a chance to learn about other classes later in this book.

The `GRect` class

The `GRect` class allows you to create rectangles and add them to the graphics window. For example, the program in Figure 3-5 creates a graphics window and then adds a rectangle to the window, solidly filled using the color blue, as shown in the following image of the graphics window:

![Blue Rectangle](image)

**FIGURE 3-5** Program to draw a blue rectangle on the graphics window

```javascript
/*
 * File: BlueRectangle.js
 * ---------------------
 * This program uses the object-oriented graphics model to draw a
 * blue rectangle on the screen.
 */

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 200;

/* Main program */
function BlueRectangle() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let rect = GRect(150, 50, 200, 100);
    rect.setColor("Blue");
    rect.setFilled(true);
    gw.add(rect);
}
```
For the most part, the BlueRectangle.js program looks much the same as the GraphicsHelloWorld.js program from Figure 3-3. It includes—as all of the graphics programs do in this book—constant definitions indicating the size of the graphics window and a main program that begins by creating a GWindow of the desired size and assigning it to the variable gw.

The next statement in the BlueRectangle function is

```javascript
let rect = GRect(150, 50, 200, 100);
```

which creates a GRect object used to display the rectangle in the window. In this call, the first two arguments, 150 and 50, indicate the x and y coordinates at which the rectangle should be positioned; the second two arguments, 200 and 100, specify the width and height of the rectangle. As in the earlier call to GWindow, each of these values is measured in pixels, but it is important to keep in mind that the coordinate values in the y direction increase as you move down the screen, with the (0, 0) origin in the upper left corner. To maintain consistency with this convention, the origin of a graphical object is usually defined to be its upper left corner. The GRect object stored in the variable rect is therefore positioned so that its upper left corner is at the point (150, 50) relative to the upper left corner of the window. This geometry is illustrated in Figure 3-6.

The remaining statements in the BlueRectangle function are all examples of method calls. For example, the statement
3.3 Classes in the graphics library

rect.setColor("Blue");

sends the rectangle object a setColor message asking it to change its color. The argument to setColor is a string representing one of the many color names that JavaScript defines, which are listed in Figure 3-7. In this case, the setColor call tells the rectangle to set its color to blue.

If the 140 standard web colors listed in Figure 3-7 are not enough for you, JavaScript allows you to specify 16,777,216 different colors by indicating the proportion of the three primary colors of light: red, green, and blue. To do so, all you need to do is specify the color as a string in the form "#rrggbb", where rr indicates the red value, gg indicates the green value, and bb indicates the blue value. Each of these values is expressed as a two-digit number written in hexadecimal, which is base 16. You may already be familiar with this form of color specification.

![Figure 3-7: Predefined color names in JavaScript](image)

<table>
<thead>
<tr>
<th>AliceBlue</th>
<th>DarkSlateGrey</th>
<th>LightPink</th>
<th>PaleVioletRed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AntiqueWhite</td>
<td>DarkTurquoise</td>
<td>LightSalmon</td>
<td>PapayaWhip</td>
</tr>
<tr>
<td>Aqua</td>
<td>DarkViolet</td>
<td>LightSeaGreen</td>
<td>PeachPuff</td>
</tr>
<tr>
<td>Aquamarine</td>
<td>DeepPink</td>
<td>LightSkyBlue</td>
<td>Peru</td>
</tr>
<tr>
<td>Azure</td>
<td>DeepSkyBlue</td>
<td>Light SlateGray</td>
<td>Pink</td>
</tr>
<tr>
<td>Beige</td>
<td>DimGrey</td>
<td>Light SlateGray</td>
<td>Plum</td>
</tr>
<tr>
<td>Bisque</td>
<td>DimGrey</td>
<td>LightSteelBlue</td>
<td>PowderBlue</td>
</tr>
<tr>
<td>Black</td>
<td>DodgerBlue</td>
<td>LightYellow</td>
<td>Purple</td>
</tr>
<tr>
<td>BlanchedAlmond</td>
<td>FireBrick</td>
<td>Lime</td>
<td>RebeccaPurple</td>
</tr>
<tr>
<td>Blue</td>
<td>FloralWhite</td>
<td>LimeGreen</td>
<td>Red</td>
</tr>
<tr>
<td>BlueViolet</td>
<td>ForestGreen</td>
<td>Linen</td>
<td>RosyBrown</td>
</tr>
<tr>
<td>Brown</td>
<td>Fuchsia</td>
<td>Magenta</td>
<td>RoyalBlue</td>
</tr>
<tr>
<td>BurlyWood</td>
<td>Gainboro</td>
<td>Maroon</td>
<td>SaddleBrown</td>
</tr>
<tr>
<td>CadetBlue</td>
<td>GhostWhite</td>
<td>MediumAquaMarine</td>
<td>Salmon</td>
</tr>
<tr>
<td>Chartreuse</td>
<td>Gold</td>
<td>MediumBlue</td>
<td>SandyBrown</td>
</tr>
<tr>
<td>Chocolate</td>
<td>GoldenRod</td>
<td>MediumOrchid</td>
<td>SeaGreen</td>
</tr>
<tr>
<td>Coral</td>
<td>Gray</td>
<td>MediumPurple</td>
<td>SeaShell</td>
</tr>
<tr>
<td>Cornsilk</td>
<td>Green</td>
<td>MediumSeaGreen</td>
<td>Silver</td>
</tr>
<tr>
<td>Crimson</td>
<td>GreenYellow</td>
<td>MediumSlateBlue</td>
<td>Silver</td>
</tr>
<tr>
<td>Cyan</td>
<td>HoneyDew</td>
<td>MediumSlateGray</td>
<td>Snow</td>
</tr>
<tr>
<td>DarkBlue</td>
<td>HotPink</td>
<td>MintCream</td>
<td>SpringGreen</td>
</tr>
<tr>
<td>DarkCyan</td>
<td>IndianRed</td>
<td>MistyRose</td>
<td>SteelBlue</td>
</tr>
<tr>
<td>DarkGoldenRod</td>
<td>Indigo</td>
<td>Moccasin</td>
<td>Tan</td>
</tr>
<tr>
<td>DarkGray</td>
<td>Ivory</td>
<td>NavajoWhite</td>
<td>Navy</td>
</tr>
<tr>
<td>DarkKhaki</td>
<td>Khaki</td>
<td>OldLace</td>
<td>Teal</td>
</tr>
<tr>
<td>DarkMagenta</td>
<td>Lavender</td>
<td>Olive</td>
<td>Thistle</td>
</tr>
<tr>
<td>DarkOliveGreen</td>
<td>LemonChiffon</td>
<td>OliveDrab</td>
<td>Tomato</td>
</tr>
<tr>
<td>DarkOrange</td>
<td>LightBlue</td>
<td>Orange</td>
<td>Turquoise</td>
</tr>
<tr>
<td>DarkOrchid</td>
<td>LightCoral</td>
<td>OrangeRed</td>
<td>Violet</td>
</tr>
<tr>
<td>DarkRed</td>
<td>LightCyan</td>
<td>Orchard</td>
<td>Wheat</td>
</tr>
<tr>
<td>DarkSalmon</td>
<td>LightGoldenRodYellow</td>
<td>Olive</td>
<td></td>
</tr>
<tr>
<td>DarkSeaGreen</td>
<td>LightGray</td>
<td>PaleGoldenRod</td>
<td>White</td>
</tr>
<tr>
<td>DarkSlateBlue</td>
<td>LightGrey</td>
<td>PaleGreen</td>
<td>WhiteSmoke</td>
</tr>
<tr>
<td>DarkSlateGray</td>
<td>LightGreen</td>
<td>PaleTurquoise</td>
<td>Yellow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>YellowGreen</td>
</tr>
</tbody>
</table>
from designing web pages. If not, you will have a chance to learn more about hexadecimal notation in Chapter 7.

The next line in the `BlueRectangle` function is the method call

```javascript
rect.setFilled(true);
```

which sends a `setFilled` message to the rectangle. In this case, the argument to the `setFilled` method is the value `true`. The values `true` and `false` are instances—and in fact the only instances—of an extremely important type in JavaScript that you will learn more about in Chapter 4. For the moment, however, it is sufficient to think of these two values in terms of their conventional interpretation. Calling `rect.setFilled(true)` indicates that the rectangle should be filled. Conversely, calling `rect.setFilled(false)` indicates that it should not be, which leaves only the outline.

By default, the `GRect` function creates rectangles that are unfilled. Thus, if you left this statement out of `BlueRectangle.js`, the result would look like this:

![Unfilled Rectangle](image)

The rectangle is still blue, but is outlined rather than filled.

The final line in the `BlueRectangle` function is the method call

```javascript
gw.add(rect);
```

which sends an `add` message to the graphics window, asking it to add the graphical object stored in `rect` to the contents of the window. The rectangle is added to the window at coordinates that have already been set at the time that the rectangle was created. Adding the rectangle produces the final contents of the display.

**The `GOval` class**

As its name suggests, the `GOval` class is used to display an oval-shaped figure in a graphics window. Structurally, the `GOval` class is similar to the `GRect` class: the `GOval` function itself takes the same arguments as the `GRect` function, and the two classes respond to the same set of methods. The difference lies in the figures those
classes produce on the screen. The GRect class displays a rectangle whose location and size are determined by the argument values $x$, $y$, width, and height. The GOval class displays the oval whose edges just touch the boundaries of that rectangle.

The relationship between the GRect and the GOval classes is most easily illustrated by example. The following function definition takes the code from the earlier BlueRectangle.js program and extends it by adding a GOval with the same coordinates and dimensions:

```javascript
function GRectPlusGOval() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let rect = GRect(150, 50, 200, 100);
    rect.setFilled(true);
    rect.setColor("Blue");
    gw.add(rect);
    let oval = GOval(150, 50, 200, 100);
    oval.setFilled(true);
    oval.setColor("Red");
    gw.add(oval);
}
```

The resulting output looks like this:

There are two important things to notice in this example. First, the red GOval extends so that its edges touch the boundary of the rectangle. Second, the GOval, which was added after the GRect, hides the portions of the rectangle that lie underneath the boundary of the oval. If you were to add these figures in the opposite order, all you would see is the blue GRect, because the entire GOval would be underneath the boundaries of the GRect.

**The GLine class**

The GLine class is used to display line segments on the graphics window. The GLine function takes four arguments, which are the $x$ and $y$ coordinates of the two endpoints. For example, the function call
GLine(0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT)

creates a GLine object running from the point (0, 0) in the upper left corner to the point at the opposite corner in the lower right. In the 500×200 graphics windows used in the all the examples so far, this line would run from (0, 0) to (500, 200).

The following function uses the GLine class to draw the two diagonals across the graphics window:

```javascript
function DrawDiagonals() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    gw.add(GLine(0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT));
    gw.add(GLine(0, GWINDOW_HEIGHT, GWINDOW_WIDTH, 0));
}
```

Loading this program into the browser generates the following display:

![DrawDiagonals](image)

The GLabel class

When you last saw the GLabel class in the GraphicsHelloWorld.js program, the results were not entirely satisfying. The message appeared on the screen, but was too small to generate much excitement. To make the "hello, world" message bigger, you need to display the GLabel in a different font.

In all likelihood, you already know about fonts from working with other computer applications and have an intuitive sense that fonts determine the style in which characters appear. More formally, a font is an encoding that maps characters into images that appear on the screen. To change the font of the GLabel, you need to send it a setFont message, which might look like this:

```javascript
msg.setFont("36px 'Times New Roman'");
```

This call to the setFont method tells the GLabel stored in msg to change its font to one in which the height of a text line is 36 pixels and the font family is Times
New Roman used by *The New York Times*. After making that call, the graphics window will look like this:

![Hello World](image)

The string passed as the argument to `setFont` is written using CSS, which, as noted earlier in this chapter, is the technology the web uses to specify the visual appearance of the page. This string specifies several properties of the font, which appear in the following order:

- The **font style**, which specifies which of several alternative forms of the font should be used. This specification is ordinarily omitted from the font string to indicate a normal font but may appear as *italic* or *oblique* to indicate an italic variant or a slanted one.

- The **font weight**, which specifies how dark the font should be. This specification is omitted for normal fonts but may appear as *bold* to specify a boldface one.

- The **font size**, which specifies how tall the characters should be by indicating the distance between two successive lines of text. In CSS, the font size is usually specified in pixel units as a number followed by the suffix `px`, as in the `36px` specification in the previous example.

- The **family name**, which indicates the name associated with the font. If the name of the font contains spaces, it must be quoted, usually using single quotation marks because the entire font specification appears as a JavaScript string. Setting the text in Times New Roman, for example, therefore requires the font string to include `'Times New Roman'` as in the previous example. Because different computers support different fonts, CSS allows a font specification to include several family names separated by commas. The browser will then use the first font family that is available. Particularly if you think a font is unlikely to exist on some computers, it is good practice to end the list with one of CSS’s **generic family names**, which do not name a specific font but describe a kind of font that is certain to be available in some form. The most common generic font names are *serif*, which indicates a font with decorative bits at the edges of the character image such as those in Times New Roman, *sans-serif*, which indicates a font lacking those decorations such as Arial or Helvetica, and *monospace*, which indicates that all characters should have the same width as in Monaco or Courier New.
As you probably know from using your word processor, it can be fun to experiment with different fonts. On most Macintosh systems, for example, there is a font called Lucida Blackletter that produces a script reminiscent of the style of illuminated manuscripts of medieval times. To set the message in this font, you could change the `setFont` call in this program to

```javascript
msg.setFont("24px 'Lucida Blackletter',serif");
```

Note that the font string includes the generic family name `serif` as an alternative. If the browser displaying the page could not find a font called Lucida Blackletter, it could then substitute one of the standard serif fonts, such as Times New Roman. If, however, it were able to load the Lucida Blackletter font successfully, the output would look something like this:

![hello, world](image)

The `GLabel` class uses its own geometric model, which is similar to the ones that typesetters have used over the centuries since Gutenberg’s invention of the printing press. The notion of a font, of course, originally comes from printing. Printers would load different sizes and styles of type into their presses to control the way in which characters appeared on a page. The terminology that the graphics library uses to describe both fonts and labels also derives from the typesetting world. You will find it easier to understand the behavior of the `GLabel` class if you learn the following terms:

- The **baseline** is the imaginary line on which characters sit.
- The **origin** is the point at which the text of a label begins. In languages that read left to right, the origin is the point on the baseline at the left edge of the first character. In languages that read right to left, the origin is the point at the right edge of the first character, at the right end of the line.
- The **height** is the distance between successive baselines in multiline text.
- The **ascent** is the maximum distance characters extend above the baseline.
- The **descent** is the maximum distance characters extend below the baseline.

The interpretation of these terms in the context of the `GLabel` class is illustrated in Figure 3-8.
The \texttt{GLabel} class includes methods that allow you to determine these properties. For example, the \texttt{GLabel} class includes a method called \texttt{getAscent} to determine the ascent of the font in which the label appears. In addition, it includes a method called \texttt{getWidth} that determines the horizontal extent of the \texttt{GLabel}.

These methods make it possible to center a label in the window, although they raise an interesting question. The only function you’ve seen to create a \texttt{GLabel} takes its initial coordinates as parameters. If you want to center a label, you won’t know those coordinates until after you have created the label. To solve this problem, the function that creates a \texttt{GLabel} comes in two forms. The first takes the string for the label along with the \texttt{x} and \texttt{y} coordinates of the origin. The second leaves out the origin point, which sets the origin to the default value of (0, 0).

Suppose, for example, that you want to center the string "hello, world" in the graphics window. To do so, you first need to create the \texttt{GLabel}, then change its font so that it has the right appearance, and finally determine the dimensions of the label to calculate the correct initial position. You can then supply those coordinates in the \texttt{add} method, which takes optional \texttt{x} and \texttt{y} parameters to set the origin of the object when you add it to the \texttt{GWindow}. The following program implements this strategy:

```javascript
function CenteredHelloWorld() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let msg = GLabel("hello, world");
    msg.setFont("36px 'Sans-Serif' ");
    let x = (gw.getWidth() - msg.getWidth()) / 2;
    let y = (gw.getHeight() + msg.getAscent()) / 2;
    gw.add(msg, x, y);
}
```

The calculations necessary to center the \texttt{GLabel} occur in the declarations of the variables \texttt{x} and \texttt{y}, which specify the origin point for the centered label. To compute
the $x$ coordinate of the label, you need to shift the origin left by half the width of the label from the center of the window. Centering the label in the vertical dimension is a bit trickier. You can get pretty close by defining the $y$ coordinate to be half the font ascent below the centerline. These declarations also introduce the fact that the GWindow object also implements the `getWidth` and `getHeight` methods, so you can use these method calls to determine the width and height of the window.

Running the `CenteredHelloWorld` function produces the following image on the graphics window:

![Hello World Image](image)

If you’re a stickler for aesthetic detail, you may find that using `getAscent` to center a `GLabel` vertically doesn’t produce the optimal result. Most labels that you display on the canvas will appear to be a few pixels too low. The reason is that `getAscent` returns the maximum ascent of the font and not the distance the text of this particular `GLabel` happens to rise above the baseline. For most fonts, certain characters—most notably the parentheses and accent marks—extend above the tops of the uppercase letters and therefore increase the font ascent. If you want things to look perfect, you may have to adjust the vertical centering by a pixel or two.

### The GWindow Class

Although it is essential for any program that uses the graphics library, the GWindow class is conceptually different from the other classes in the package. Classes like `GRect` and `GLabel` represent objects that you can display in a graphics window. The GWindow class represents the graphics window itself.

The GWindow object is conventionally initialized by the line

```javascript
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
```

which appears at the beginning of every program that uses the graphics package. This statement creates the graphics window and installs it in the web page so that it is visible to the user. It also serves to implement the conceptual framework for displaying graphical objects. The conceptual framework implemented by a library package is called its model. The model gives you a sense of how you should think about working with that package.
One of the most important roles of a model is to establish what analogies and metaphors are appropriate for the package. Many real-world metaphors are possible for computer graphics, just as there are many different ways to create visual art. One possible metaphor is that of painting, in which the artist selects a paintbrush and a color and then draws images by moving the brush across a screen that represents a virtual canvas.

For consistency with the principles of object-oriented design, the Stanford Graphics library uses the metaphor of a collage. A collage artist works by taking various objects and assembling them on a background canvas. In the real world, those objects might be geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. The graphics library offers counterparts for all these objects.

The fact that the graphics window uses the collage model has implications for the way you describe the process of creating a design. If you are painting, you might talk about making a brush stroke in a particular position or filling an area with paint. In the collage model, the key operations are adding and removing objects, along with repositioning them on the background canvas.

Collages also have the property that some objects can be positioned on top of other objects, obscuring whatever is behind them. Removing those objects reveals whatever used to be underneath. The back-to-front ordering of objects in the collage is called the stacking order in this book, although you will sometimes see it referred to as z-ordering in more formal writing. The name z-ordering comes from the fact that the stacking order occurs along the axis that comes out of the two-dimensional plane formed by the x and y axes. In mathematics, the axis coming out of the plane is called the z-axis.

The most important methods in the GWindow class appear in Figure 3-9. Other classes and methods will be introduced in later chapters as they become relevant.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GWindow(width, height)</td>
<td>Creates a new GWindow object of the specified size.</td>
</tr>
<tr>
<td>gw.add(obj)</td>
<td>Adds the object to the graphics window at its internally stored location.</td>
</tr>
<tr>
<td>gw.add(obj, x, y)</td>
<td>Adds the object to the graphics window so that its origin is positioned at the point (x, y).</td>
</tr>
<tr>
<td>gw.remove(obj)</td>
<td>Removes the object from the graphics window.</td>
</tr>
<tr>
<td>gw.getWidth()</td>
<td>Returns the width of the graphics window.</td>
</tr>
<tr>
<td>gw.getHeight()</td>
<td>Returns the height of the graphics window.</td>
</tr>
</tbody>
</table>
3.4 Class hierarchies

One of the defining properties of languages like JavaScript that support the object-oriented paradigm is that they allow you to define hierarchical relationships among classes. Those hierarchies are similar in many ways to the biological classification system developed by the eighteenth-century Swedish botanist Carl Linnaeus as a means of representing the structure of the biological world. In Linnaeus’s conception, living things are first subdivided into kingdoms. Each kingdom is further broken down into the hierarchical categories of phylum, class, order, family, genus, and species. Every living species belongs not only to its own category at the bottom of the hierarchy but also to a category at each higher level.

This biological classification system is illustrated in Figure 3-10, which shows the classification of the common black garden ant, whose scientific name,
Lasius niger, corresponds to its genus and species. This species of ant, however, is also part of the family Formicidae, which is the classification that actually identifies it as an ant. If you move upward in the hierarchy from there, you discover that Lasius niger is also of the order Hymenoptera (which includes bees and wasps), the class Insecta (which consists of the insects), and the phylum Arthropoda (which includes, for example, shellfish and spiders).

One of the properties that makes this biological classification system useful is that all living things belong to a category at every level in the hierarchy. Each individual life form therefore belongs to several categories simultaneously and inherits the properties that are characteristic of each one. The species Lasius niger, for example, is an ant, an insect, an arthropod, and an animal—all at the same time. Moreover, each individual ant shares the properties that it inherits from each of those categories. One of the defining characteristics of the class Insecta is that insects have six legs. All ants must therefore have six legs because ants are members of that class.

The biological metaphor also helps to illustrate the distinction between classes and objects. Although every common black garden ant has the same biological classification, there are many individuals of the common-black-garden-ant variety. In the language of object-oriented programming, Lasius niger is a class and each individual ant is an object.

Class structures in JavaScript follow much the same hierarchical pattern, as illustrated in Figure 3-11, which shows the relationships among the classes in the graphics library described in this chapter. The GWindow class is in a category by itself. The other class at the top of the diagram is GObject, which is the class that

**Figure 3-11** Simplified UML diagram for the class in the Stanford graphics library

- **GWindow**
  - add(obj)
  - add(obj, x, y)
  - remove(obj)
  - getWidth()
  - getHeight()

- **GObject**
  - getX()
  - getY()
  - getExtWidth()
  - getHeight()
  - setColor(color)

- **GRect**
  - GRect(x, y, width, height)
  - GRect(width, height)
  - setFilled(flag)
  - setFillColor(color)

- **GOval**
  - GOval(x, y, width, height)
  - setFilled(flag)
  - setFillColor(color)

- **GLine**
  - GLine(x1, y1, x2, y2)
  - setFont(str)

- **GLabel**
  - GLabel(str, x, y)
Running Programs in the Browser

encompasses every graphical object that can be displayed in a GWindow. The diagram in Figure 3-11 adopts parts of a standard methodology for illustrating class hierarchies called the Universal Modeling Language, or UML for short. In UML, each class appears as a rectangular box whose upper portion contains the name of the class. The methods implemented by that class appear in the lower portion; these methods are described in more detail in Figure 3-12.

---

**FIGURE 3-12 Selected methods from the Stanford Graphics Library**

**Factory methods to create graphical objects**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GRect(x, y, width, height)</code></td>
<td>Creates a <code>GRect</code> object with the specified dimensions.</td>
</tr>
<tr>
<td><code>GRect(width, height)</code></td>
<td>Creates a <code>GRect</code> object of the specified size with its origin at (0, 0).</td>
</tr>
<tr>
<td><code>GOval(x, y, width, height)</code></td>
<td>Creates a <code>GOval</code> that fits inside the bounds of the corresponding rectangle.</td>
</tr>
<tr>
<td><code>GOval(width, height)</code></td>
<td>Creates a <code>GOval</code> object in which the oval fits inside a rectangle of the specified size. The origin of the <code>GOval</code> is (0, 0).</td>
</tr>
<tr>
<td><code>GLine(x1, y1, x2, y2)</code></td>
<td>Creates a <code>GLine</code> object connecting <code>(x1, y1)</code> and <code>(x2, y2)</code>.</td>
</tr>
<tr>
<td><code>GLabel(str, x, y)</code></td>
<td>Creates a <code>GLabel</code> object containing the specified string with its baseline origin at the point <code>(x, y)</code>.</td>
</tr>
<tr>
<td><code>GLabel(str)</code></td>
<td>Creates a <code>GLabel</code> object containing the specified string with its baseline origin at the point (0, 0).</td>
</tr>
</tbody>
</table>

**Methods common to all graphical objects**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object.getX()</code></td>
<td>Returns the x coordinate of the object.</td>
</tr>
<tr>
<td><code>object.getY()</code></td>
<td>Returns the y coordinate of the object.</td>
</tr>
<tr>
<td><code>object.getWidth()</code></td>
<td>Returns the width of the graphical object.</td>
</tr>
<tr>
<td><code>object.getHeight()</code></td>
<td>Returns the height of the graphical object.</td>
</tr>
<tr>
<td><code>object.setColor(color)</code></td>
<td>Sets the color of the object to <code>color</code>.</td>
</tr>
</tbody>
</table>

**Methods available only for the GRect and GOval classes**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object.setFill(flag)</code></td>
<td>Sets whether this object is filled.</td>
</tr>
<tr>
<td><code>object.setFillColor(color)</code></td>
<td>Sets the color used to fill the interior of the object.</td>
</tr>
</tbody>
</table>

**Methods available only for the GLabel class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object.setFont(str)</code></td>
<td>Sets the font for the label. The format of the font specification is a CSS string as described in the text.</td>
</tr>
<tr>
<td><code>object.getAscent()</code></td>
<td>Gets the ascent (maximum distance above the baseline) for the label font.</td>
</tr>
<tr>
<td><code>object.getDescent()</code></td>
<td>Gets the descent (maximum distance below the baseline) for the label font.</td>
</tr>
</tbody>
</table>
UML diagrams use open arrowheads to point from one class to the class at a higher level of the hierarchy from which it inherits its behavior. The class that appears lower in the hierarchy is a subclass of the class to which it points, which is called its superclass. In the UML diagram in Figure 3-11, the name of the GObject class appears in italics. This notation is used to define an abstract class, which is a class that is never used to create an object but instead acts as a common superclass for concrete classes that appear beneath it in the hierarchy. Because GObject is abstract, you never create a GObject but instead create one of its concrete subclasses, which in this diagram are GRect, GOval, GLine, and GLabel.

All subclasses of GObject inherit its methods, so that every GRect, GOval, GLine, and GLabel implements the setColor method, which is common to all graphical objects. The subclasses, however, often implement additional methods that are particular to that subclass. For example, both GRect and GOval implement the methods setFilled and setFillColor, which determine how the interior of the shape is colored. A GLine object, however, has no interior, so the entire concept of filling makes no sense in that context. Similarly, the idea of a font applies only to the GLabel class, which means that the setFont method is defined for that class and not at some higher

### 3.5 Functions that return graphical objects

It is important to keep in mind that graphical objects are data values in JavaScript in precisely the same way that numbers and strings are. You can therefore assign graphical objects to variables, pass them as arguments to function calls, or have functions return them as results. Figure 3-13 illustrates this feature by defining a function createFilledCircle that takes four arguments: the values x and y representing the coordinates of the center of the circle, a number r specifying the radius of the circle, and a string color indicating the JavaScript color name. The Target function itself calls createFilledCircle three times to create three circles that alternate in color between red and white and that progressively decrease in size, producing the following output:
Although you may sometimes get lucky with extremely simple programs, one of the truths you will soon have to accept as a programmer is that very few of your programs will run correctly the first time around. Most of the time, you will need to spend a considerable fraction of your time testing the program to see whether it works, discovering that it doesn’t, and then settling into the process of debugging, in which you find and fix the errors in your code.
3.6 Testing and debugging

Perhaps the most compelling description of the centrality of debugging to the programming process comes from the British computing pioneer Maurice Wilkes (1913–2010), who in 1979 offered the following reflection from his early years in the field:

As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

**Becoming a good debugger**

Debugging is one of the most creative and intellectually challenging aspects of programming. It can, however, also be one of the most frustrating. If you are just beginning your study of programming, it is likely that the frustrating aspects of debugging will loom much larger than the excitement of overcoming an interesting intellectual challenge. That fact in itself is by no means surprising. Debugging, after all, is a skill that takes time to learn. Before you have developed the necessary experience and expertise, your forays into the world of debugging will often leave you facing a completely mysterious problem that you have absolutely no idea how to solve. And when your assignment is due the next day and you can make no progress until you somehow solve that mystery, frustration is probably the most natural reaction.

To a surprising extent, the problems that people face while debugging are not so much technical as they are psychological. To become a successful debugger, the most important thing is to start thinking in new ways that gets you beyond the psychological barriers that stand in your way. There is no magical, step-by-step approach to finding the problems, which are almost always of your own making. What you need is logic, creativity, patience, and a considerable amount of practice.

**The phases of the programming process**

When you are developing a program, the actual process of writing the code is only one piece of a more complex intellectual activity. Before you sit down to write the code, it is almost always wise to spend some time thinking about the program design. As you discovered when you were working with Karel in Chapter 1, there are usually many ways to decompose a large problem into more manageable pieces. Putting some thought into the design of that decomposition before you start writing the individual functions is almost certain to reduce the total amount of time—and frustration—involved in the project as a whole. After you’ve written the code, you need to test whether it works and, in all probability, spend some time ferreting out the bugs that prevent the program from doing what you want.
These four activities—designing, coding, testing, and debugging—constitute the principal components of the programming process. And although there are certainly some constraints on order (you can’t debug code that you haven’t yet written, for example), it is a terrible mistake to think of these phases as rigidly sequential. The biggest problem that students have comes from thinking that it makes sense to design and code the entire program and then try to get it working as a whole. Professional programmers would never do it that way. They develop a preliminary design, write some pieces of the code, test those pieces to see if they work as intended, and then fix the bugs that the testing uncovers. Only when that individual piece is working do professional programmers return to code, test, and debug the next section of the program. From time to time, professional programmers will go back and revisit the design as they learn from the experience of seeing how well the original design works in practice. You have to learn to work in much the same way.

It is equally important to recognize that each phase in the programming process requires a fundamentally different approach. As you move back and forth among the different phases, you need to adopt different ways of thinking. In my experience, the best way to illustrate how these approaches differ is to associate each phase with a profession that depends on much the same skills and modes of thought.

During the design phase, you need to think like an architect. You need to have a sense not only of the problem that needs to be solved but also an understanding of the underlying aesthetics of different solution strategies. Those aesthetic judgments are not entirely free from constraints. You know what’s needed, you recognize what’s possible, and you choose the best design that lies within those constraints.

When you move to the coding phase, your role shifts to that of the engineer. Your job is to apply your understanding of programming to transform a theoretical design into an actual implementation. This phase is by no means mechanical and requires a significant amount of creativity, but your goal is to produce a program that you believe implements the design.

In many respects, the testing phase is the most difficult aspect of the process to understand. When you act as a tester, your role is not to establish that the program works. It is in fact just the opposite. Your job is to break it. A tester therefore needs to operate in the role of a vandal. You need to search deliberately for anything that might go wrong and take real joy in finding any flaws. It is precisely in this role that the most difficult psychological barriers arise. As the coder, you want the program to work; as the tester, you want it to fail. Many people have trouble shifting focus in this way. After all, it’s hard to be overjoyed at pointing out the stupid mistakes the coder made when you also happen to be that coder. Even so, you need to make this shift.
Finally, your job as the debugger is that of a detective. The testing process reveals the existence of errors but does not necessarily reveal why they occur. Your job during the debugging phase is to sort through all the available evidence, create a hypothesis about what is going wrong, verify that hypothesis through additional testing, and then make the necessary corrections.

As with testing, the debugging phase is full of psychological pitfalls when you act in the detective role. When you were writing the code in your role as engineer, you believed that it did what you intended it to do when you designed it in your role as architect. You now have to discover why it doesn’t, which means that you have to discard any preconceptions you’ve retained from those earlier phases and come at the problem with a fresh perspective. Making that shift successfully is always a difficult challenge. Code that looked correct to you once is likely to look just as good when you come back to it a second time.

What you need to keep in mind is that the testing phase determined that the program is not working correctly. There must be a problem somewhere. It’s not the browser or JavaScript that’s misbehaving or some unfortunate conjunction of the planets. As Cassius reminds Brutus in Shakespeare’s Julius Caesar, “the fault, dear Brutus, is not in our stars, but in ourselves.” You introduced the error when you wrote the code, and it is your responsibility to find it.

This book will offer additional suggestions about debugging as you learn how to write more complex program, but the following principle will serve you better than any specific debugging strategy or technique:

When you are trying to find a bug, it is more important to understand what your program is doing than to understand what it isn’t doing.

Most people who come upon a problem in their code go back to the original problem and try to figure out why their program isn’t doing what they wanted. Such an approach can be helpful in some cases, but it is far more likely that this kind of thinking will make you blind to the real problem. If you make an unwarranted assumption the first time around, you are likely to make it again, and be left in the position that you can’t see any reason why your program isn’t doing the right thing. What you need to do instead is gather information about what your program is in fact doing and then try to work out where it goes wrong.

Although many modern browsers come equipped with sophisticated JavaScript debuggers, you are likely to get the most mileage out of the console.log function. If you discover that your program isn’t working, add a few calls to console.log at places where you think your program might be going down the wrong path. In some cases, it is sufficient to include a line like

```javascript
console.log("I got here");
```
to the program. If the message "I got here" appears on the console, you know that the program got to that point in the code. It is often even more helpful to have the call to `console.log` display the value of an important variable. If, for example, you expect the variable `n` to have the value 100 at some point in the code, you can add the line

```javascript
console.log("n = " + n);
```

If running the program shows that `n` has the value 0 instead, you know that something has gone wrong prior to this point. Narrowing down the region of the program in which the problem might be located puts you in a much better position to find and correct the error.

Since the process of debugging is so similar to the art of detection, it seems appropriate to offer some of the more relevant bits of debugging wisdom I’ve encountered in detective fiction, which appear in Figure 3-14. I also recommend strongly Robert Pirsig’s critically acclaimed novel *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values* (Bantam, 1974), which stands as the best exposition of the art and psychology of debugging ever written. The most relevant section is the discussion of “gumption traps” in Chapter 26.

![Debugging advice from detective fiction](image)

<table>
<thead>
<tr>
<th>Figure 3-14</th>
<th>Debugging advice from detective fiction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regard with distrust all circumstances which seem to favor our secret desires.</td>
<td>—Émile Gaboriau, <em>Monstre Lecoq</em>, 1868</td>
</tr>
<tr>
<td>There is nothing like first-hand evidence.</td>
<td>—Sir Arthur Conan Doyle, <em>A Study in Scarlet</em>, 1888</td>
</tr>
<tr>
<td>It is a capital mistake to theorise before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.</td>
<td>—Sir Arthur Conan Doyle, <em>A Scandal in Boヘния</em>, 1892</td>
</tr>
<tr>
<td>It is of the highest importance in the art of detection to be able to recognize out of a number of facts which are incidental and which vital. Otherwise your energy and attention must be dissipated instead of being concentrated.</td>
<td>—Sir Arthur Conan Doyle, <em>The Adventure of the Reigate Squires</em>, 1892</td>
</tr>
<tr>
<td>With method and logic one can accomplish anything.</td>
<td>—Agatha Christie, <em>Poirot Investigates</em>, 1924</td>
</tr>
<tr>
<td>Detection requires a patient persistence which amounts to obstinacy.</td>
<td>—P. D. James, <em>An Unsuitable Job for a Woman</em>, 1972</td>
</tr>
<tr>
<td>It was always more difficult than you thought it would be.</td>
<td>—Alexander McCall Smith, <em>The No. 1 Ladies' Detective Agency</em>, 1998</td>
</tr>
</tbody>
</table>
An example of a psychological barrier

Although most testing and debugging challenges involve a level of programming sophistication beyond the scope of this chapter, there is a very simple program that illustrates just how easy it is to let your assumptions blind you not only to the cause of an error but even to its very existence. Throughout the many years that I have taught computer science, one of my favorite problems to assign at the beginning of the term is to write a function that solves the quadratic equation

\[ ax^2 + bx + c = 0 \]

As you know from high school, this equation has two solutions given by the formula

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

The first solution is obtained by using + in place of the \( \pm \) symbol; the second is obtained by using – instead. The problem I assign to students is to write a function that takes \( a, b, \) and \( c \) as parameters and displays the two resulting solutions for \( x \). The students are allowed to assume that \( a \) is not equal to zero and that the value under the square-root sign is nonnegative, which guarantees that real solutions exist.

Although a majority of students are able to solve this problem correctly, there are always a significant number—as much as 20 percent of a large class—who turn in functions that look something like this:

```javascript
function quadratic(a, b, c) {
    let root = Math.sqrt(b*b - 4*a*c);
    let x1 = (-b + root) / 2*a;
    let x2 = (-b - root) / 2*a;
    console.log("x1 = " + x1);
    console.log("x2 = " + x2);
}
```

As the bug symbol indicates, this implementation of `quadratic` is incorrect, although the problem is subtle. It looks as if the expression \( 2*a \) is in the denominator of the fraction, when in fact it isn’t. In JavaScript, operators in the same precedence class, such as the / and * in the lines defining \( x1 \) and \( x2 \), are evaluated in left-to-right order, which means that the parenthesesized value in these expressions is first divided by 2 and then multiplied by \( a \). The quadratic formula requires the denominator to be the quantity \( (2*a) \), which means that the parentheses are necessary.

The real lesson from this example, however, lies in the fact that many students compound their mistake by failing to discover it. Most of the students who make
this error fail to test their programs for any values of the coefficient $a$ other than 1, since those are the easiest answers to compute by hand. If $a$ is 1, it doesn’t matter whether you multiply or divide by $a$ because the answer will be the same. Worse still, students who test their program for other values of $a$ often fail to notice that their programs give incorrect answers. I often get sample runs that look like this:

```
> quadratic(8, -6, 1)
x1 = 32
x2 = 16
>
```

This sample run asserts that $x = 32$ and $x = 16$ are solutions to the equation

$$8x^2 - 6x + 1 = 0$$

but it is easy to check that neither of these values in fact satisfy the equation. Even so, students happily submit programs that generate this sample run without noticing that the answers are wrong.

**Writing effective test programs**

Whenever you write a function, it is a good idea to write a companion function that checks that your implementation works in a large set of cases. Figure 3-15 on the next page shows how a test program for the `quadratic` function can be included in the program file along with the function definition. This test program generates several test runs of the `quadratic` functions with a range of parameters. Moreover, to make sure that anyone running the program doesn’t simply believe the answers coming out of the computer, the program indicates exactly what the correct answers should be. A complete sample run of the `TestQuadratic` program looks like this:

```
> TestQuadratic()
x^2 + 5x + 6 = 0 (roots should be -2 and -3):
x1 = -2
x2 = -3

x^2 + x - 12 = 0 (roots should be 3 and -4):
x1 = 3
x2 = -4

x^2 - 10x + 25 = 0 (roots should be 5 and 5):
x1 = 5
x2 = 5

8x^2 - 6x + 1 = 0 (roots should be 0.5 and 0.25):
x1 = 0.5
x2 = 0.25
>
```
Although it is impossible to test all possible inputs for a function, it is usually possible to identify a set of test cases that check for the most likely sources of error. For the quadratic function, for example, your test function should make sure to check a range of values for the coefficients \(a\), \(b\), and \(c\). It is also important, as the example from the previous section demonstrates, to know what the answers should be.

Each of the sample programs supplied with this book contains a test function of this sort, and it is good practice for you to adopt this approach in your own code. Thinking about testing as you write the program will make it much easier to find the bugs that will inevitably show up in your code from time to time.

### Summary

In this chapter, you learned how to create the necessary JavaScript and HTML files to run JavaScript programs on the web. The classic “Hello World” program in

```javascript
/*
 * File: Quadratic.js
 * ------------------
 * This file defines the quadratic function, which solves the quadratic
 * equation given the coefficients \(a\), \(b\), and \(c\).
 */

function quadratic(a, b, c) {
    let root = Math.sqrt(b*b - 4*a*c);
    let x1 = (-b + root) / (2*a);
    let x2 = (-b - root) / (2*a);
    console.log("x1 = "+ x1);
    console.log("x2 = "+ x2);
}

/* Simple program to test the quadratic function */

function TestQuadratic() {
    console.log("x^2 + 5x + 6 = 0 (roots should be -2 and -3): ");
    quadratic(1, 5, 6);
    console.log(" ");
    console.log("x^2 + x - 12 = 0 (roots should be 3 and -4): ");
    quadratic(1, 1, -12);
    console.log(" ");
    console.log("x^2 - 10x + 25 = 0 (roots should be 5 and 5): ");
    quadratic(1, -10, 25);
    console.log(" ");
    console.log("8x^2 - 6x + 1 = 0 (roots should be 0.5 and 0.25): ");
    quadratic(8, -6, 1);
}
Figure 3-1 illustrates how to write a program that uses the `console.log` method to display output in the console window. Subsequent examples show you how to create graphical programs that draw rectangles, ovals, lines, and strings on the graphics window.

Important points introduced in the chapter include:

- In the defining document for the programming language C, Brian Kernighan and Dennis Ritchie suggest that the first program written in any language should be one that prints the string "hello, world". The advantage of running such a simple program is that doing so teaches you all the other things you need to know about writing programs in that language. This book follows their advice.

- JavaScript was designed for use in conjunction with the World Wide Web. For this reason, JavaScript programs ordinarily run in the context of a browser displaying a web page rather than as standalone applications.

- Modern web pages use three distinct technologies to define the contents of a web page. The structure and contents of the page are defined using a file written using HTML (Hypertext Markup Language), the visual appearance of the page is specified using CSS (Cascading Style Sheets), and the interactive behavior is defined using JavaScript.

- Every JavaScript program that runs in a browser must include an `index.html` file that defines the overall structure of the page, loads the necessary JavaScript programs and libraries, and specifies a JavaScript expression to be evaluated when the page is loaded. These `index.html` files have a conventional form, which appears in Figure 3-2 for programs that use the console and Figure 3-4 for programs that use graphics.

- JavaScript files are loaded into the browser by means of `<script>` tags in the `index.html` file. These tags have the following form:

  ```html
  <script type="text/javascript" src="filename"></script>
  ```

  where `filename` indicates the name of the file.

- The graphical programs in this book use the Stanford Graphics Library, which is a collection of graphical tools designed for use in introductory courses. That library is supplied as a single JavaScript file called `JSGraphics.js`.

- JavaScript supports a modern style of programming called the object-oriented paradigm, which focuses attention on data objects and their interactions.

- In the object-oriented paradigm, an object is a conceptually integrated entity that ties together the information that defines the state of the object and the operations that affect its state. Each object is a representative of a class, which is a template that defines the attributes and operations shared by all objects of a
particular type. A single class can give rise to many different objects; each such object is an instance of that class.

- Objects communicate by sending messages. In JavaScript, those messages are implemented by calling methods, which are simply functions that belong to a particular class.

- Method calls in JavaScript use the receiver syntax, which looks like this:

  \[ \text{receiver}. \text{name} (\text{arguments}) \]

  The receiver is the object to which the message is sent, name indicates the name of the method that responds to the message, and arguments is a list of values that convey any additional information carried by the message.

- Functions that create new objects are called factory methods and conventionally have names that begin with an uppercase letter.

- The first line in any JavaScript program that uses the Stanford Graphics Library is to create a GWindow object using the following declaration:

  \[ \text{let gw} = \text{GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)}; \]

  The constants GWINDOW_WIDTH and GWINDOW_HEIGHT appear earlier in the program and specify the dimensions of the window in pixels, which are the tiny dots that cover the face of the display.

- Once the variable gw has been initialized, the rest of a graphics program creates graphical objects of various kinds and adds them to the window to create the desired graphical image.

- This chapter introduces four classes of graphical objects—GRect, GOval, GLine, and GLabel—that represent rectangles, ovals, line segments, and text strings, respectively. Other GObject subclasses are introduced in later chapters.

- All of the graphical object classes support the method setColor, which takes the name of the color as a string. JavaScript defines 140 standard colors whose names appear in Figure 3-7 on page 75.

- The GRect and GOval classes use the setFilled and setFillColor methods to control whether the interior of the shape is filled and the color used for the interior.

- The GLabel class uses the setFont methods to set the font in which the label appears. The argument to setFont is the CSS specification of a font, which is described on page 79.

- The GLabel class uses a geometric model that is different from the one used by the other graphical objects. That model is illustrated in Figure 3-8 on page 81.

- Classes in JavaScript form hierarchies similar to the classification system used in biology.
• In a JavaScript class hierarchy, subclasses automatically have access to the methods defined in their superclasses. This behavior is called inheritance.

• The classes in the Stanford Graphics Library form a hierarchy in which every graphical object is a subclass of GObject, which defines the methods common to all graphical objects, such as setColor. Methods that apply only to specific classes are defined at a lower level in the hierarchy.

• The four phases of the programming process are design, coding, testing, and debugging, although it is best to view these phases as interrelated rather than sequential. Professional programmers typically code one piece of a program, test it, debug it, and then go back and work on the next piece.

• Each phase in the programming process requires you to behave in a different way. During the design phase, you act as an architect. When you are coding, you are acting as an engineer. During testing, it is important to imagine yourself as a vandal, given that your goal is to break the program, not to prove that it works. When you are debugging, you need to think like a detective and call in all the cleverness and insight of a Sherlock Holmes.

• When you are trying to find a bug, it is more important to understand what your program is doing than to understand what it isn’t doing.

• In seeking to understand what your program is doing, your most helpful resource is the console.log function.

• The most serious problems programmers face during the testing and debugging phases are psychological rather than technical. It is extremely easy to let your assumptions and desires get in the way of understanding where the problems lie.

• One of the most insightful references on the psychology and philosophy of debugging in Robert Pirsig’s Zen and the Art of Motorcycle Maintenance: An Inquiry into Values (Bantam, 1974).

• It is good programming practice to include test programs along with the definitions of any functions that you write.

**Review questions**

1. What did Brian Kernighan and Dennis Ritchie suggest should be the first program you write in any language? What reasons did they offer for starting off with a program that simple?

2. What are the three technologies used to specify a web page? What aspects of the web page do each of these technologies control?

3. What is the conventional name of the HTML file that defines a web page?
4. What is the syntax of the HTML tag used to load JavaScript files into the browser?

5. What is the name of the JavaScript library used in this chapter to implement programs that write output to the console? What reasons does the chapter give for using this library to replace the standard system console?

6. What is the name of the JavaScript library used in this chapter to implement programs that produce graphical output?

7. In your own words, define the terms class, object, and method.

8. The object-oriented paradigm uses the metaphor of sending messages to model communication between objects. How does JavaScript implement this idea?

9. What is the receiver syntax?

10. What is a factory method?

11. What is the first line in every function used in this text to implement a graphical program?

12. What are the four classes of graphical objects introduced in this chapter?

13. How do you change the color of a graphical object?

14. What is the purpose of the setFilled and setFillColor methods in the GRect and GOval classes?

15. What is the format of the argument string passed to setFont?

16. Define the following terms in the context of the GLabel class: baseline, origin, height, ascent, and descent.

17. Explain the purpose of the two following lines in the CenteredHelloWorld function:

   ```javascript
   let x = (gw.getWidth() - msg.getWidth()) / 2;
   let y = (gw.getHeight() + msg.getAscent()) / 2;
   ```

18. When you center a GLabel vertically using the getAscent method, why does the resulting text often appear to be a few pixels too low?

19. What is the collage model?

20. What is meant by the term stacking order? What other term does the chapter suggest is often used for the same purpose?
21. Define the following terms: subclass, superclass, and inheritance.

22. Why does the class name GObject appear in italics in Figure 3-11?

23. Why does it make sense to implement setColor but not setFilled as part of the GObject class?

24. What are the four phases of the programming process identified in this chapter? For each of those phases, what professional role does the chapter offer as a model for how to approach that phase?

25. True or false: Professional programmers work through the four phases of the programming process sequentially, finishing each one before moving on to the next.

26. True or false: When you are testing your program, your primary goal is to show that it works.

27. What piece of advice does the chapter offer to help you think effectively about debugging?

28. What built-in JavaScript function does the chapter identify as the most useful debugging tool?

### Exercises

1. Use an editor to create the program HelloWorld.js and the file index.html exactly as they appear in Figures 3-1 and 3-2. Use your browser to open the index.html file to show that you can get JavaScript programs working.

2. Do the same for the GraphicsHelloWorld.js program and the associated index.html from Figures 3-3 and 3-4.

3. Write a graphical program TicTacToeBoard.js that draws a Tic-Tac-Toe board centered in the graphics window, as shown in the following sample run:
The size of the board should be specified as a constant, and the diagram should be centered in the window, both horizontally and vertically.

4. Use the graphics library to draw a rainbow that looks something like this:

![Rainbow](image)

Starting at the top, the seven bands in the rainbow are red, orange, yellow, green, blue, indigo, and violet, respectively; cyan makes a lovely color for the sky. Remember that this chapter defines only the `GRect`, `GOval`, `GLine`, and `GLabel` classes and does not include a graphical object that represents an arc. It will help to think outside the box, in a more literal sense than usual.

5. Draw a simplified version of Figure 3-8, which illustrates the geometry of the `GLabel` class. In your implementation, you should display the two strings ("The quick brown fox" and "jumped over the lazy dog") in red using a sans-serif font that is large enough to make the guidelines easy to see. You should then for each of the strings draw a gray line along the baseline, the line that marks the font ascent, and the line that marks the font descent. Finally, you should draw a small filled circle indicating the baseline origin of the first string. The graphics window will then look like this:

![GLabelGeometry](image)

Note that this output is a little more honest than Figure 3-8 about the font ascent, which appears slightly above the top of the uppercase characters. If the strings included characters like parentheses or accent marks, some of these would extend all the way to the ascent line.
6. Write a JavaScript program that draws a simplified diagram of the `GObject` hierarchy, as follows:

The only classes you need to create this picture are `GRect`, `GLabel`, and `GLine`. The tricky part is specifying the coordinates so that the different elements of the picture are aligned properly. The aspects of the alignment for which you are responsible are:

- The width and height of the class boxes must be specified as constants.
- The labels should be centered in their boxes.
- The connecting lines should start and end at the center of the appropriate edge of the box.
- The entire figure should be centered in the window.

7. For each of the exercises in Chapter 2, create an `index.html` file that loads the corresponding JavaScript files and displays the result of running a suitable test program in the context of a web page.
I had a running compiler and nobody would touch it. . . . They carefully told me, computers could only do arithmetic; they could not do programs.


Grace Murray Hopper studied mathematics and physics at Vassar College and went on to earn her Ph.D. in mathematics at Yale. During the Second World War, Hopper joined the United States Navy and was posted to the Bureau of Ordinance Computation at Harvard University, where she worked with computing pioneer Howard Aiken. Hopper became one of the first programmers of the Mark I digital computer, which was one of the first machines capable of performing complex calculations. Hopper made several contributions to computing in its early years and was one of the major contributors to the development of the language COBOL, which continues to have widespread use in business programming applications. In 1985, Hopper became the first woman promoted to the rank of admiral. During her life, Grace Murray Hopper served as the most visible example of a successful woman in computer science. In recognition of that contribution, there is now a biennial Celebration of Women in Computing, named in her honor.
In Chapters 2 and 3, you saw several examples of simple JavaScript functions, some of which used the console to display output and some of which produced graphical images. Every one of these functions, however, started at the first statement in its body and then executed each of the remaining statements in order, possibly calling some functions along the way. Before you can write more interesting applications, you need to learn how to control the operation of your program in more sophisticated ways, much as you did with the Karel programs in Chapter 1.

Just like their counterparts in Karel, functions in JavaScript often include control statements to determine the order of operation. The if and the while statements are essentially the same in both languages, but JavaScript uses a more flexible statement called for to achieve the effect of the repeat statement in Karel. In addition, JavaScript includes a conditional statement called switch that makes it easier to write code that needs to choose among several possible execution paths. This chapter covers these control statements in JavaScript and, in the process, extends the set of tools you have for solving problems.

Students often believe that there must be some rule that determines when they need to use each of the various control statements that a programming language provides. That’s not how programming works. Control statements are tools for solving problems. Before you can determine what control statement makes sense in a particular context, you have to give serious thought to the problem you are trying to solve and the strategy you have chosen to solve it. You write the code for a program after you have decided how to solve the underlying problem. There is nothing automatic about the programming process.

Although students are sometimes disappointed to learn that there are no magic rules that turn a problem statement into a working program, that fact is what makes programming such a valuable skill. If it were possible to carry out the programming process according to some well-defined algorithm, it would then be straightforward to automate the entire process and eliminate the need for programmers altogether. Programming requires an enormous amount of ingenuity and creativity. The essence of programming consists of solving problems, many of which are extremely complex and challenging. Solving such problems is what makes computer programming hard; it is also what makes programming interesting and fun.

4.1 Boolean data

The major difference between control statements in the two languages lies in the conditions you can check. Karel’s conditional expressions, which have names like frontIsClear and facingNorth, make sense only in Karel’s world. In JavaScript, you express conditions by constructing expressions whose values are either true or false. Such expressions are called Boolean expressions, after the
4.1 Boolean data

Mathematician George Boole, who developed an algebraic approach for working with this type of data. Boolean values are represented in JavaScript using a built-in type whose domain consists of exactly two values: `true` and `false`.

JavaScript defines several operators that work with Boolean values. These operators fall into two classes—relational operators and logical operators—which are discussed in the next two sections.

**Relational operators**

The simplest questions that you can ask in JavaScript are those that compare two data values. You might want, for example, to determine whether two values are equal or if one is greater than or less than another. Traditional mathematics uses the operators `=`, `≠`, `<`, `>`, `≤`, and `≥` to signify the relationships *equal to*, *not equal to*, *less than*, *greater than*, *less than or equal*, and *greater than or equal*, respectively. Unfortunately, several of these symbols don’t appear on a standard keyboard, which means that JavaScript programs represent these operators in a slightly different form that uses the following character combinations in place of the conventional mathematical symbols:

- `===` Equal to
- `!==` Not equal to
- `<` Less than
- `>` Greater than
- `<=` Less than or equal to
- `>=` Greater than or equal to

Collectively, these operators are called *relational operators* because they test the relationship between two values. Like the arithmetic operators introduced in Chapter 2, relational operators appear between the two values to which they apply. For example, if you need to check whether the value of `x` is less than 0, you can use the expression `x < 0`.

At first glance, the relational operators `===` and `!==` probably seem a bit strange. Because the single equal sign had already been reserved to indicate assignment, the designers of the C programming language from which JavaScript is derived introduced a new operator consisting of two adjacent equal signs to specify equality. The designers of JavaScript retained the `==` operator, but defined it in such a confusing way that it is hard for anyone—novices and experienced programmers alike—to use it correctly. In JavaScript, the operators that check for exact equality and exact inequality are `===` and `!==`, and you will save yourself a great deal of confusion if you use these operators in preference to the shorter forms.
Logical operators

In addition to the relational operators, which take atomic values of any type and produce Boolean results, JavaScript defines three operators that take Boolean operands and combine them to form other Boolean values:

- ! Logical not (true if the following operand is false)
- && Logical and (true if both operands are true)
- || Logical or (true if either or both operands are true)

These operators are called *logical operators* and are listed in decreasing order of precedence.

The operators &&, ||, and ! resemble the English words *and*, *or*, and *not*. Even so, it is important to remember that English is somewhat imprecise when it comes to logic. To avoid that imprecision, it is helpful to think of these operators in a more formal, mathematical way. Logicians define these operators using *truth tables*, which show how the value of a Boolean expression changes as the values of its operands change. For example, the truth table for the && operator, given Boolean values \( p \) and \( q \), is

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( p &amp;&amp; q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

The last column of the table indicates the value of the Boolean expression \( p && q \) given the individual values of the Boolean variables \( p \) and \( q \) shown in the first two columns. Thus, the first line in the truth table shows that when \( p \) is false and \( q \) is false, the value of the expression \( p && q \) is also false.

The truth table for || is

| \( p \)  | \( q \)  | \( p || q \) |
|--------|--------|-------------|
| false  | false  | false       |
| false  | true   | true        |
| true   | false  | true        |
| true   | true   | true        |

Even though the || operator corresponds to the English word *or*, it does not indicate *one or the other*, as it often does in English, but instead indicates *either or both*, which is its mathematical meaning.
The `!` operator has the following simple truth table:

<table>
<thead>
<tr>
<th>p</th>
<th>!p</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

If you need to determine how a more complex logical expression operates, you can break it down into these primitive operations and build up a truth table for the individual pieces of the expression.

In most cases, logical expressions are not so complicated that you need a truth table to figure them out. The only case that often causes confusion is when the `!` operator comes up in conjunction with `&&` or `||`. When English speakers talk about situations that are not true (as is the case when you work with the `!` operator), a statement whose meaning is clear to human listeners is often at odds with mathematical logic. Whenever you find that you need to express a condition involving the word `not`, you should use extra care to avoid errors.

As an example, suppose you wanted to express the idea “x is not equal to either 2 or 3” as part of a program. Just reading from the English version of this conditional test, new programmers are very likely to code this expression as follows:

```javascript
x !== 2 || x !== 3
```

As noted in Chapter 1, this book uses the bug symbol to mark sections of code that contain deliberate errors. In this case, the problem is that an informal English translation of the code does not correspond to its interpretation in JavaScript. If you look at this conditional test from a mathematical point of view, you can see that the expression is `true` if either (a) `x` is not equal to 2 or (b) `x` is not equal to 3. No matter what value `x` has, one of the statements must be `true`, since, if `x` is 2, it cannot also be equal to 3, and vice versa. To fix this problem, you need to refine your understanding of the English expression so that it states the condition more precisely. That is, you want the condition to be `true` whenever “it is not the case that either `x` is 2 or `x` is 3.” You could translate this expression directly to JavaScript by writing

```javascript
!(x === 2 || x === 3)
```

but the resulting expression is a bit ungainly. The question you really want to ask is whether both of the following conditions are `true`:

- `x` is not equal to 2, and
- `x` is not equal to 3.
If you think about the question in this form, you can write the test as

\[ x \neq 2 \&\& x \neq 3 \]

This simplification is a specific illustration of the following more general relationship from mathematical logic:

\[ \neg (p \lor q) \text{ is equivalent to } \neg p \&\& \neg q \]

for any logical expressions \( p \) and \( q \). This transformation rule and its symmetric counterpart

\[ \neg (p \&\& q) \text{ is equivalent to } \neg p \lor \neg q \]

are called De Morgan’s laws after the British mathematician Augustus De Morgan. Forgetting to apply these rules and relying instead on the English style of logic is a common source of programming errors.

**Short-circuit evaluation**

JavaScript interprets the \&\& and || operators in a way that differs from the interpretation used in many other programming languages. In the programming language Pascal, for example, evaluating these operators (which are written as AND and OR) requires evaluating both halves of the condition, even when the result can be determined partway through the process.

The designers of JavaScript (or, more accurately, the designers of the languages on which JavaScript is based) took a different approach that is usually more convenient for programmers. Whenever JavaScript evaluates an expression of the form

\[ exp_1 \&\& exp_2 \]

or

\[ exp_1 \lor \lor exp_2 \]

the individual subexpressions are always evaluated from left to right, and evaluation ends as soon as the answer can be determined. For example, if \( exp_1 \) is false in the expression involving \&\& , there is no need to evaluate \( exp_2 \) since the final answer will always be false. Similarly, in the example using \lor , there is no need to evaluate the second operand if the first operand is true. This style of evaluation, which stops as soon as the answer is known, is called short-circuit evaluation.

A primary advantage of short-circuit evaluation is that it allows one condition to control the execution of a second one. In many situations, the second part of a
compound condition is meaningful only if the first part comes out a certain way. For example, suppose you want to express the combined condition that (1) the value of the integer $x$ is nonzero and (2) $x$ divides evenly into $y$. You can express this conditional test in JavaScript as

$$(x !== 0) \land (y \% x === 0)$$

because the expression $y \% x$ is evaluated only if $x$ is nonzero. The corresponding expression in Pascal fails to generate the desired result, because both parts of the Pascal condition will always be evaluated. Thus, if $x$ is 0, a Pascal program containing this expression will end up dividing by 0 even though it appears to have a conditional test to check for that case. Conditions that protect against evaluation errors in subsequent parts of a compound condition, such as the conditional test

$$(x !== 0)$$

in the preceding example, are called guards.

### 4.2 The if statement

The simplest way to express conditional execution in JavaScript is by using the `if` statement, which comes in the same two forms that you saw in Karel, which are shown in the syntax boxes on the right. The condition component of these templates is a Boolean expression, as defined in the preceding section. In the simple form of the `if` statement, JavaScript executes the block of statements only if the conditional test evaluates to `true`. If the conditional test is `false`, JavaScript skips the body of the `if` statement entirely. In the form that includes the `else` keyword, JavaScript executes the first block of statements if the condition is `true` and the second if the condition is `false`. In both forms of the `if` statement, the code that JavaScript executes when the conditional expression is `true` is called the `then clause`. In the `if-else` form, the code executed when the condition is `false` is called the `else clause`.

You can use the `if` statement to implement several of the simplest functions in JavaScript’s `Math` class. For example, you can implement `abs` as follows:

```javascript
function abs(x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```
Similarly, you can implement \texttt{max}—at least for two arguments—like this:

```javascript
function max(x, y) {
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

As you almost certainly realized during your study of Karel, the choice of whether to use the \texttt{if} or the \texttt{if-else} form depends on the structure of the problem. You use the simple \texttt{if} statement when your problem requires code to be executed only if a particular condition applies. You use the \texttt{if-else} form for situations in which the program must choose between two independent sets of actions. You can often make this decision based on how you would describe the problem in English. If that description contains the word \textit{otherwise} or some similar word, there is a good chance that you’ll need the \texttt{if-else} form. If the English description conveys no such notion, the simple form of the \texttt{if} statement is probably sufficient.

### Additional formats for the \texttt{if} statement

If either of the clauses associated with the \texttt{if} consists of a single statement, JavaScript allows you to eliminate the curly braces that surround it. As a general rule, it is good practice to include these braces because doing so makes it easier for someone reading the code to determine which statements are included in the body. Requiring the braces also makes programs easier to maintain, by making it harder for someone to imagine that two statements are governed by an \texttt{if} statement when the braces are omitted. For example, it is easy to misread the lines

```javascript
if (condition)
  statement1;
  statement2;
```

and think that the \texttt{if} statement controls the execution of both statements. Without the braces, \texttt{statement2} is outside the range of the \texttt{if} and will always be executed.

There are two situations in which the programs in this book relax the rule that every clause in an \texttt{if} statement requires curly braces. The first of these is when an \texttt{if} statement fits on a single line and has no \texttt{else} clause. The second is when the \texttt{else} clause consists of another test to check for some additional condition. Such statements are called \textit{cascading if statements} and may involve any number of \texttt{else if} lines. For example, the following function implements the \texttt{sign} function from the \texttt{Math} class, which returns $-1$, $0$, or $+1$ depending on whether the value of \texttt{x} is positive, zero, or negative:

```javascript
if (x > 0) {
  return 1;
} else if (x < 0) {
  return -1;
} else {
  return 0;
}
```
function sign(x) {
    if (x < 0) {
        return -1;
    } else if (x === 0) {
        return 0;
    } else {
        return 1;
    }
}

Note that there is no need to check explicitly for the $x > 0$ condition. If the program reaches that last else clause, there is no other possibility, since the earlier tests have eliminated the negative and zero cases.

In many situations, it makes more sense to use the switch statement to choose among a set of independent cases than to adopt the cascading if form. The switch statement is described in section 4.3.

The ?: operator

The JavaScript programming language provides another, more compact mechanism for expressing conditional execution that can be extremely useful in certain situations: the ?: operator. (This operator is referred to as question-mark colon, even though the two characters do not actually appear adjacent to each other.) Unlike any other operator in JavaScript, ?: requires three operands. The general form of the operation is

$$condition \ ? \ expression_1 : expression_2$$

When JavaScript encounters the ?: operator, it first evaluates the condition. If the condition turns out to be true, $expression_1$ is evaluated and used as the value of the entire expression; if the condition is false, the value is the result of evaluating $expression_2$. The ?: operator is therefore a space-efficient form of the following idea

if (condition) {
    Use the value of $expression_1$
} else {
    Use the value of $expression_2$
}

implemented in the context of an expression.

For example, you can use the ?: operator to implement the function max in the following streamlined form:
The parentheses around the condition are not technically required, but many JavaScript programmers include them in this context to enhance the readability of the code.

4.3 The `switch` statement

The `if` statement is ideal for applications in which the program logic calls for a two-way decision point: some condition is either `true` or `false`, and the program acts accordingly. Some applications, however, call for more complicated decision structures involving more than two choices, where those choices can be divided into a set of mutually exclusive cases: in one case, the program should do `x`; in another case, it should do `y`; in a third, it should do `z`; and so forth. In many applications, the most appropriate statement to use for such situations is the `switch` statement, which is outlined in the syntax box on the left.

The header line of the `switch` statement is

```
switch (expression) {
    case c1:
        statements
        break;
    case c2:
        statements
        break;
    case c3:
        statements
        break;
    default:
        statements
        break;
}
```

where `e` is an expression. In the context of the `switch` statement, this expression is called the **control expression**. The body of the `switch` statement is divided into individual groups of statements introduced with one of two keywords: `case` or `default`. A `case` line and all the statements that follow it up to the next instance of either of these keywords are called a **case clause**. The `default` line and its associated statements are called the **default clause**. For example, in the template shown in the syntax box, the range of statements

```
case c1:
    statements
    break;
```

constitutes the first `case` clause.

When the program executes a `switch` statement, the control expression `e` is evaluated and compared against the values `c1`, `c2`, and so forth, each of which must be a constant. If one of the constants matches the value of the control expression, the statements in the associated `case` clause are executed. When the program reaches the `break` statement at the end of the clause, the operations specified by that clause are complete, and the program continues with the statement following

```javascript
function max(x, y) {
    return (x > y) ? x : y;
}
```
the entire `switch` statement. If none of the case constants match the value of the control expression, the statements in the `default` clause are executed.

The template shown in the syntax box deliberately suggests that the `break` statements are a required part of the syntax. I encourage you to think of the `switch` syntax in precisely that form. JavaScript is defined so that if the `break` statement is missing, the program starts executing statements from the next clause after it finishes the selected one. While this design can be useful in some cases, it tends to cause more problems than it solves. To reinforce the importance of remembering to include the `break` statement, every `case` clause in this text ends with an explicit `break` or a `return` statement.

The one exception to this rule is that multiple `case` lines specifying different constants can appear together, one after another, before the same statement group. For example, a `switch` statement might include the following code:

```javascript
    case 1:
    case 2:
        statements
        break;
```

which indicates that the specified statements should be executed if the `select` expression is either 1 or 2. The JavaScript interpreter treats this construction as two `case` clauses, the first of which is empty. Because the empty clause contains no `break` statement, a program that selects that path simply continues on with the second clause. From a conceptual point of view, however, you are probably better off thinking of this construction as a single `case` clause that represents two possibilities.

The `default` clause is optional in the `switch` statement. If none of the cases match and there is no `default` clause, the program simply continues on with the next statement after the `switch` statement without taking any action at all. To avoid the possibility that the program might ignore an unexpected case, it is good programming practice to include a `default` clause in every `switch` statement unless you are certain you have enumerated all the possibilities.

Because `switch` statements can be rather long, programs are easier to read if the `case` clauses themselves are short. If there is room to do so, it also helps to put the `case` identifier, the statements forming the body of the clause, and the `break` or `return` statement all together on the same line. This style is illustrated in the `monthName` function in Figure 4-1, which uses a `switch` statement to translate a numeric month into its name.
The default clause in Figure 4-1 deserves special mention. If the month parameter is not one of the legal values between 1 and 12, the `monthName` function returns the predefined JavaScript constant `undefined`, which is often used as a marker to indicate that no meaningful result exists.

### 4.4 The while statement

The simplest iterative construct is the `while` statement, which repeatedly executes a simple statement or block until the conditional expression becomes `false`. The template for the `while` statement appears in the syntax box on the left. The entire statement, including both the `while` control line itself and the statements enclosed within the body, constitutes a **while loop**. When the program executes a `while` statement, it first evaluates the conditional expression to see if it is `true` or `false`. If the condition is `false`, the loop **terminates** and the program continues with the next statement after the entire loop. If the condition is `true`, the entire body is executed, after which the program goes back to the top to check the condition again. A single pass through the statements in the body constitutes a **cycle** of the loop.
There are two important principles to observe about the operation of a `while` loop:

1. The conditional test is performed before every cycle of the loop, including the first. If the test is `false` initially, the body of the loop is not executed at all.
2. The conditional test is performed only at the beginning of a loop cycle. If that condition happens to become `false` at some point during the loop, the program doesn’t notice that fact until it has executed a complete cycle. At that point, the program evaluates the test condition again. If it is still `false`, the loop terminates.

**Using the while loop**

Learning how to use the `while` loop effectively usually requires looking at several examples in which the `while` loop comes up naturally in the solution strategy. One application that has particular historical significance is that of finding the largest proper divisor of an integer, which is the first program run on the first modern computer, the Small-Scale Experimental Machine at Manchester University, which was nicknamed the “Baby.” The author of the program was Alan Turing, one of the great pioneers of computer science whose name is now remembered in computer science’s highest honor, the annual Turing Award.

Because the Baby had extremely limited capabilities, Turing’s algorithm had to be almost absurdly simple. Given a number \( N \), Turing’s program simply counted down from \( N - 1 \) until it found a number that divided evenly into \( N \). Taking at least some advantage of JavaScript’s extended set of operations, Turing’s algorithm might look like this:

```javascript
function largestFactor(n) {
  let factor = n - 1;
  while (n % factor !== 0) {
    factor--;
  }
  return factor;
}
```

Two sample calculations using `largestFactor` appear in the following sample run:

<table>
<thead>
<tr>
<th>LargestFactor</th>
</tr>
</thead>
</table>
| > largestFactor(63)  
21                |
| > largestFactor(262144)  
131072          |
| >                     |
The second of these calculations is the one run on the Manchester Baby on June 21, 1948. The program took 52 minutes to compute the answer. In the process, it demonstrated both the efficacy and the reliability of the Baby’s architecture.

As a second example, suppose that you have been asked to write a function \texttt{digitSum} that adds up the digits in a positive integer. Calling \texttt{digitSum(1729)} should therefore produce the result 19, which is $1 + 7 + 2 + 9$. How would you go about implementing such a function?

The first thing that your function needs to do is keep track of a running total. The usual strategy for doing so is to declare a variable called \texttt{sum}, initialize it to 0, add each digit to \texttt{sum} one at a time, and finally return the value of \texttt{sum}. That much of the structure, with the rest of the problem written in English, is shown below:

\begin{verbatim}
function digitSum(n) {
    let sum = 0;
    For each digit in the number, add that digit to \texttt{sum}.
    return sum;
}
\end{verbatim}

Programs that are written partly in a programming language and partly in English are called \textit{pseudocode}.

The sentence

\begin{quote}
For each digit in the number, add that digit to \texttt{sum}.
\end{quote}

clearly specifies a loop structure of some sort, since there is an operation that needs to be repeated for each digit in the number. If it were easy to determine how many digits a number contained, you might choose to use the \texttt{for} loop described later in this chapter to run through precisely that many cycles. As it happens, finding out how many digits there are in a number is just as hard as adding them up in the first place. The best way to write this program is just to keep adding in digits until you discover that you have added the last one. Loops that run until some condition occurs are most often coded using the \texttt{while} statement.

The essence of this problem lies in determining how to break up a number into its component digits. The last digit of an integer \texttt{n} is simply the remainder left over when \texttt{n} is divided by 10, which is the result of the expression \texttt{n \% 10}. The rest of the number—the integer that consists of all digits except the last one—is given by \texttt{Math.floor(n / 10)}, which is denoted in mathematics as $\lfloor n / 10 \rfloor$. For example, if \texttt{n} has the value 1729, you can use these two expressions to break that number into two parts, 172 and 9, as shown in the following diagram:
Thus, in order to add up the digits in the number, all you need to do is add the value \( n \% 10 \) to the variable `sum` on each cycle of the loop and then replace the value of \( n \) by `Math.floor(n / 10)`. The next cycle will add in the second-to-last digit from the original number, and so on, until all the digits have been processed.

But how do you know when to stop? As you compute `Math.floor(n / 10)` in each cycle, you will eventually reach the point at which \( n \) becomes 0. At that point, you’ve processed all the digits in the number and can exit from the loop. In other words, as long as the value of \( n \) is greater than 0, you should keep going. Thus, the `while` loop needed for the problem is

```javascript
while (n > 0) {
    sum += n % 10;
    n = Math.floor(n / 10);
}
```

The full implementation of the `digitSum` function appears in Figure 4-2.

---

**Figure 4-2** Function to add up the digits in a number

```javascript
/*
* File: DigitSum.js
* ---------------
* The file defines the function digitSum, which sums the digits in a number.
*/

/*
* Returns the sum of the digits in n, which must be a nonnegative integer.
*/

function digitSum(n) {
    let sum = 0;
    while (n > 0) {
        sum += n % 10;
        n = Math.floor(n / 10);
    }
    return sum;
}
```
As a final example of the use of `while`, it is often useful to be able to add spaces to a string to ensure that strings of different lengths still line up correctly when displayed in a fixed-width font of the sort used in the JavaScript console. For example, columns of numbers are conventionally aligned on the right, which means that it is necessary to add spaces at the beginning of a number until it fills the entire width of the column.

You can achieve this goal with the following function, which takes a value (which can be of any type) and a field width:

```javascript
function alignRight(value, width) {
  let str = "" + value;
  while (str.length < width) {
    str = " " + str;
  }
  return str;
}
```

The function returns a string in which `value` appears at the right edge of a field that is `width` characters wide. The first line of the function uses the concatenation operator to convert `value` to a string and then adds it to the end of a string that contains no characters at all, which is called the empty string. The effect is therefore to initialize the variable `str` so that it contains the string representation of `value`. From here, the function uses concatenation to add spaces to the beginning of `str` until it has the desired length and then returns the padded string to the caller. You will have a chance to see `alignRight` in action in the following section.

### 4.5 The `for` statement

One of the most important control statements in JavaScript is the `for` statement, which is most often used in situations in which you want to repeat an operation a particular number of times. The general form of the `for` statement is shown in the syntax box on the right.

The operation of the `for` loop is determined by the three italicized expressions on the `for` control line: `init`, `test`, and `step`. The `init` expression indicates how the `for` loop should be initialized, which typically consists of a variable declaration specifying an initial value. In a `for` loop, this variable is called the index variable. The `test` expression is a conditional test written exactly like the test in a `while` statement. As long as the test expression is `true`, the loop continues. The `step` expression indicates how the index variable changes at the end of each cycle.

The interpretation of the `init`, `test`, and `step` expressions is easiest to illustrate by example. The most common `for` loop uses the following header line:
for (let i = 0; i < n; i++)

The loop begins by declaring the index variable \( i \) and initializing it to 0. The loop increments the value of \( n \) at the end of each cycle and continues as long as \( i \) is less than \( n \). The loop therefore runs for a total of \( n \) cycles, with \( i \) taking on the values 0, 1, 2, and so forth, up to the final value \( n - 1 \).

More generally, the for loop idiom

\[
\text{for (int } i = \text{start}; i <= \text{finish}; i++)
\]

starts by setting the value of \( i \) to \( \text{start} \) and then continues as long as the value of \( i \) is less than or equal to \( \text{finish} \). This loop therefore uses the variable \( i \) to count from \( \text{start} \) to \( \text{finish} \).

You can use this form of the for loop idiom to define a function called fact that takes an integer \( n \) and returns its factorial, which is defined as the product of the integers between 1 and \( n \) and traditionally written as \( n! \) in mathematics. The first several factorials are shown in the following table:

\[
\begin{align*}
0! &= 1 & \text{(by definition)} \\
1! &= 1 & = 1 \\
2! &= 2 & = 1 \times 2 \\
3! &= 6 & = 1 \times 2 \times 3 \\
4! &= 24 & = 1 \times 2 \times 3 \times 4 \\
5! &= 120 & = 1 \times 2 \times 3 \times 4 \times 5 \\
6! &= 720 & = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \\
7! &= 5040 & = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \\
8! &= 40320 & = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \\
9! &= 362880 & = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \\
10! &= 3628800 & = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10 \\
\end{align*}
\]

Factorials have extensive applications in statistics, combinatorial mathematics, and computer science. A function to compute factorials is therefore a useful tool for solving problems in those domains.

As a programming problem, computing a factorial has some similarities to the problem of adding a series of numbers, which appears in the implementation of the digitSum function in Figure 4-2. That function used a variable called sum to keep track of a running total of the digits. For the factorial function, the situation is much the same, except that you have to keep track of a product rather than a sum. Keeping track of a running product uses very similar code, and the only significant difference—other than replacing the + with the * operator—is that you have to initialize the running product to 1 instead of 0. The complete implementation of the fact function therefore looks like this:
function fact(n) {
    let result = 1;
    for (let i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

In its current form, you can use the `fact` function in the console window or in other functions. It would, however, also be useful if you could generate a table of factorials such as the one shown on the preceding page. To do so, the simplest approach is to use the JavaScript console to display the output by calling the function `console.log`, which displays its argument on the console and then moves the cursor to the beginning of the next line. The `FactorialTable.js` program in Figure 4-3 displays a list of factorials that extends through the range given by the constants `LOWER_LIMIT` and `UPPER_LIMIT`, as illustrated by the following sample run:

Note that the `FactorialTable` program uses `alignRight` to display the values in fixed-width columns.

Although the `step` component of the `for` loop usually increments the index variable, that is not the only possibility. You can, for example, count by twos by replacing `i++` with `i+=2` or count backwards using `i--`.

As an illustration of counting in the reverse direction, the following function counts down from an initial value until it hits 0:

```javascript
function countdown(start) {
    for (let t = start; t >= 0; t--) {
        console.log(t);
    }
}
```
Figure 4.3: Program to display a list of factorials on the console

/*
 * File: FactorialTable.js
 * ________________
 * The program defines the function FactorialTable, which prints a table
 * of factorials in a specified range.
 */

/* Constants */

const LOWER_LIMIT = 0;
const UPPER_LIMIT = 10;
const NUMBER_WIDTH = 2;
const FACTORIAL_WIDTH = 7;

/*
 * Displays a table of factorials in the range specified by the constants
 * LOWER_LIMIT and UPPER_LIMIT.
 */

function FactorialTable() {
    for (let i = LOWER_LIMIT; i <= UPPER_LIMIT; i++) {
        console.log(alignRight(i, NUMBER_WIDTH) + "! = " +
                     alignRight(fact(i), FACTORIAL_WIDTH));
    }
}

/*
 * Returns the factorial of n. The factorial is simply the product of
 * the integers between 1 and n, inclusive.
 */

function fact(n) {
    let result = 1;
    for (let i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

/*
 * Returns a string in which value appears at the right edge of a field
 * that is width characters wide. If the string representation of value
 * contains too many characters, the returned value will be longer than
 * the specified width.
 */

function alignRight(value, width) {
    let str = " " + value;
    while (str.length < width) {
        str = " " + str;
    }
    return str;
}
Calling `countdown(10)` produces the following output on the console:

```
Countdown
10
9
8
7
6
5
4
3
2
1
0
```

The `Countdown` program demonstrates that any variable can be used as an index variable. In this case, the variable is called `t`, presumably because that is the traditional variable for a rocket countdown, as in the phrase “T minus 10 seconds and counting.”

The expressions `init`, `test`, and `step` in the `for` loop pattern are all optional, but the semicolons must appear. If `init` is missing, no initialization is performed. If `test` is missing, it is assumed to be `true`. If `step` is missing, no action occurs at the end of the loop cycle. Thus the control line

```
for (;;)
```

is identical in operation to

```
while (true)
```

**The relationship between for and while**

As it happens, the `for` statement

```
for (init; test; step) {
    statements;
}
```

is identical in operation to the `while` statement

```
init;
while (test) {
    statements;
    step;
}
```

Even though you can easily rewrite the `for` statement using `while`, there are considerable advantages to using the `for` statement whenever it is possible to do so. With a `for` statement, all the information you need to understand exactly which
cycles will be executed is contained in the header line of the statement. For example, whenever you see the statement

```javascript
for (let i = 0; i < 10; i++) {
    ...body...
}
```

in a program, you know that the statements in the body of the loop will be executed 10 times, once for each of the values of `i` between 0 and 9. In the equivalent `while` loop form

```javascript
let i = 0;
while (i < 10) {
    ...body...
    i++;
}
```

the increment operation at the bottom of the loop can easily get lost if the body is large.

**Nested for statements**

In many applications, you will discover that you need to write one `for` loop inside another so that the statements in the innermost loop are executed for every possible combination of values of the `for` loop indices. Suppose, for example, that you want to display a multiplication table showing the product of every pair of numbers in the range 1 to 10. You would like the output of the program to look like this:

<table>
<thead>
<tr>
<th>MultiplicationTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  2  3  4  5  6  7  8  9  10</td>
</tr>
<tr>
<td>2  4  6  8 10 12 14 16 18 20</td>
</tr>
<tr>
<td>3  6  9 12 15 18 21 24 27 30</td>
</tr>
<tr>
<td>4  8 12 16 20 24 28 32 36 40</td>
</tr>
<tr>
<td>5 10 15 20 25 30 35 40 45 50</td>
</tr>
<tr>
<td>6 12 18 24 30 36 42 48 54 60</td>
</tr>
<tr>
<td>7 14 21 28 35 42 49 56 63 70</td>
</tr>
<tr>
<td>8 16 24 32 40 48 56 64 72 80</td>
</tr>
<tr>
<td>9 18 27 36 45 54 63 72 81 90</td>
</tr>
<tr>
<td>10 20 30 40 50 60 70 80 90 100</td>
</tr>
</tbody>
</table>

The table consists of ten horizontal rows and ten vertical columns. To create the individual entries, you need a pair of nested `for` loops: an outer loop that runs through each of the rows and an inner loop that runs through each of the entries in each row. The code inside the inner `for` loop will be executed once for every row and column, for a total of 100 individual entries in the table.

The code to draw this multiplication table appears in Figure 4-4. The outer loop runs through each value of `i` from 1 to 10 and is responsible for displaying one row of the table on each cycle. To do so, the code first declares the variable `line` and
initializes it to be the empty string. The inner loop then runs through the values of \( j \) from 1 to 10 and concatenates the product of \( i \) and \( j \) to the end of \( line \), once again using the `alignRight` function to ensure that the columns have the same width. When the inner loop is complete, the program calls `console.log` to display the completed line of the multiplication table.

Nested loops also arise frequently in graphical applications that require you to repeat some operation in both the \( x \) and \( y \) directions. Consider, for example, how you might write a program to generate a checkerboard pattern that looks like this:
A program to generate this figure appears in Figure 4-5. It is worth taking some time to go through the code, paying particular attention to the following details:

- The program is designed so that you can easily change the dimensions of the checkerboard by changing the values of the constants N_ROWS and N_COLUMNS.

- The checkerboard is arranged so that it is centered in the graphics window. The variables x0 and y0 are used to hold the coordinates of the upper left corner of the centered board.

- The decision to fill a square is made by checking whether the sum of its row number and column number is even or odd. For white squares, this sum is even; for black squares, this sum is odd. Note, however, that you don’t need to include an if statement in the code to test this condition. All you need to do is call the setFilled method with the appropriate Boolean value.
A useful way to get some practice using nested for loops is to write programs that draw patterns on the console by displaying lines of characters. As a simple example, the following function draws a triangle in which the number of stars increases by one in each row:

```javascript
function drawConsoleTriangle(size) {
  for (let i = 1; i <= size; i++) {
    let line = "";
    for (let j = 0; j < i; j++) {
      line += "*";
    }
    console.log(line);
  }
}
```

Calling `drawConsoleTriangle(10)`, for example, produces the following output on the console:

```
* 
** 
*** 
**** 
***** 
****** 
******* 
******** 
********* 
**********
```

### 4.6 Algorithmic programming

The concept of an algorithm is fundamental to computer science. As you know from Chapter 1, the word algorithm comes from the name of the 9th-century Persian mathematician Muhammad ibn Mūsā al-Khwārizmī, whose work had significant impact on modern mathematics. Figure 4-6 contains a photograph of a statue of al-Khwārizmī near his birthplace in what is now Uzbekistan.

Although it is usually sufficient to think of an algorithm as a strategy for solving a problem, modern computer science formalizes that definition so that an *algorithm* refers to a solution strategy that meets the following conditions:

- *Clear and unambiguous*, in the sense that it is stated in an understandable form.
- *Effective*, in the sense that it is possible to carry out the steps in the strategy.
- *Finite*, in the sense that the strategy terminates after some number of steps.
The next few sections offer several examples of how control statements can be used to implement several historically important algorithms.

**An early square-root algorithm**

The use of algorithms extends much further back in history than al-Khwārizmī’s time. Almost 4000 years ago, Babylonian mathematicians used an algorithmic process to calculate square roots. The primary evidence of the existence of an algorithmic process comes from cuneiform tablets such as the one shown in Figure 4-7, which shows an approximation of the square root of 2 that is far more accurate than anyone could possibly derive through measurement alone. And although the precise details of how Babylonian mathematicians performed the necessary calculations have been lost, historians believe that their technique was similar to the algorithm described by the 1st-century Greek mathematician Hero of Alexandria. The algorithm Hero described is usually called the *Babylonian method* after its most likely origin.

The Babylonian method for calculating square roots is an example of a general technique called *successive approximation*, in which you begin by making a rough guess at the answer and then improve that guess through a series of refinements that get closer and closer to the exact answer. For example, if you want to find the square root of some number \( n \), you start by choosing some smaller number \( g \) as your first guess. At every point in the process, your guess \( g \) will be smaller or larger than
the actual square root. In either case, if you divide $n$ by $g$, the result will inevitably lie on the opposite side of the desired value. For example, if $g$ is too small, $n$ divided by $g$ will be too large, and vice versa. Averaging the two values will always give a better approximation. At each step, you simply replace your previous guess $g$ by the result of the following formula, which averages $g$ and $n$ divided by $g$:

$$\frac{g + \frac{n}{g}}{2}$$

You then continue to apply this formula to each new guess until the answer is as close to the actual value as you need it to be.

To get more of a sense of how the Babylonian method works, it helps to consider a simple example. Suppose that you want to calculate, as the scribes who incised the cuneiform tablet did, the square root of 2. One possible first guess for $g$ is 1, which is half the value of $n$. The first approximation step therefore computes the following average:

$$\frac{1 + \frac{2}{1}}{2} = \frac{3}{2} = 1.5$$
The value 1.5 is closer to the actual square root of 2—which is approximately 1.4142136—so the process is on the right track.

To calculate the next approximation, all you need to do is plug $\frac{3}{2}$ into the formula as the next value of $g$, and calculate the new average, as follows:

$$\frac{\frac{3}{2} + \frac{2}{3}}{2} = \frac{17}{12} \approx 1.416667$$

From here, you simply repeat the calculation with $\frac{17}{12}$ as the new value of $g$:

$$\frac{\frac{17}{12} + \frac{2}{\frac{17}{12}}}{2} = \frac{577}{408} \approx 1.4142157$$

Applying successive approximation one more time gives

$$\frac{\frac{577}{408} + \frac{2}{\frac{577}{408}}}{2} = \frac{665857}{470832} \approx 1.4142136$$

After just four cycles, the Babylonian method has produced an approximation to the square root of 2 that is correct to eight decimal digits. Moreover, because each step generates an approximation that is closer to the exact value, you can repeat the process to produce an approximation with any desired level of accuracy.

Figure 4-8 shows the definition of a \texttt{sqrt} function that uses the Babylonian method to approximate the square root of its argument. The function uses a \texttt{while} loop to continue the process until the approximation reaches the desired level of precision. In this implementation, the \texttt{while} loop continues until the difference between the square of the current approximation and the original number is no larger than the value of the constant \texttt{TOLERANCE}.

**Finding the greatest common divisor**

Although you have seen a few simple algorithms implemented in the context of the programming examples, you have had little chance to focus on the nature of the algorithmic process itself. Most of the programming problems you have seen so far are simple enough that the appropriate solution technique springs immediately to mind. As problems become more complex, however, their solutions require more thought, and you will need to consider more than one strategy before writing the final program.

As an illustration of how algorithmic strategies take shape, the sections that follow consider two solutions to another problem from classical mathematics, which
Figure 4-8: JavaScript program to compute square roots using the Babylonian algorithm

```javascript
/*
 * File: BabylonianSquareRoot.js
 * ____________________________
 * This file implements a function sqrt that calculates square roots
 * using the Babylonian method.
 */

/* Define a constant specifying how close the value needs to be */
const TOLERANCE = 0.000000000000001;

/*
 * Calculates the square root of n using the Babylonian method, which
 * operates as follows:
 * 1. Choose a guess g (any value will do; this code uses n / 2).
 * 2. Compute a new guess by averaging g and n / g.
 * 3. Repeat step 2 until the error is less than the desired tolerance.
 */
function sqrt(n) {
    let g = n / 2;
    while (Math.abs(n - g * g) > TOLERANCE) {
        g = (g + n / g) / 2;
    }
    return g;
}
```

is to find the greatest common divisor of two integers. Given two integers \(x\) and \(y\), the greatest common divisor (or gcd for short) is the largest integer that divides evenly into both. For example, the gcd of 49 and 35 is 7, the gcd of 6 and 18 is 6, and the gcd of 32 and 33 is 1.

Suppose that you have been asked to write a function that accepts the integers \(x\) and \(y\) as input and returns their greatest common divisor. From the caller’s point of view, what you want is a function \(\text{gcd}(x, y)\) that takes two integers as arguments and returns another integer that is their greatest common divisor. The header line for this function is therefore

```
fuction gcd(x, y)
```

In many ways, the most obvious approach to calculating the gcd is simply to try every possibility. To start, you simply “guess” that \(\text{gcd}(x, y)\) is the smaller of \(x\) and \(y\), because any larger value could not possibly divide evenly into a smaller number. You then proceed by dividing \(x\) and \(y\) by your guess and seeing if it divides evenly into both. If it does, you have the answer; if not, you subtract 1 from
your guess and try again. A strategy that tries every possibility is often called a *brute-force approach*.

The brute-force approach to calculating the `gcd` function looks like this in JavaScript:

```javascript
function gcd(x, y) {
    let guess = Math.min(x, y);
    while (x % guess !== 0 || y % guess !== 0) {
        guess--;
    }
    return guess;
}
```

Before you decide that this implementation is in fact a valid algorithm for computing the `gcd` function, you need to ask yourself several questions about the code. Will the brute-force implementation of `gcd` always give the correct answer? Will it always terminate, or might the function continue forever?

To see that the program gives the correct answer, you need to look at the condition in the `while` loop

```
x % guess !== 0 || y % guess !== 0
```

As always, the `while` condition indicates under what circumstances the loop will continue. To find out what condition causes the loop to terminate, you have to negate the `while` condition. Negating a condition involving `&&` or `||` can be tricky unless you remember how to apply De Morgan’s law, which was introduced in the section on “Logical operators” earlier in this chapter. De Morgan’s law indicates that the following condition must hold when the `while` loop exits:

```
x % guess === 0 && y % guess === 0
```

From this condition, you can see immediately that the final value of `guess` is certainly a common divisor. To recognize that it is in fact the greatest common divisor, you have to think about the strategy embodied in the `while` loop. The critical factor to notice in the strategy is that the program counts *backward* through all the possibilities. The greatest common divisor can never be larger than `x` (or `y`, for that matter), and the brute-force search therefore begins with that value. If the program ever gets out of the `while` loop, it must have already tried each value between `x` and the current value of `guess`. Thus, if there were a larger value that divided evenly into both `x` and `y`, the program would already have found it in an earlier iteration of the `while` loop.
To recognize that the function terminates, the key insight is that the value of \texttt{guess} must eventually reach 1, unless a larger common divisor is found. At this point, the \texttt{while} loop will surely terminate, because 1 will divide evenly into both \texttt{x} and \texttt{y}, no matter what values those variables have.

\textbf{Euclid’s algorithm}

Brute force is not, however, the only effective strategy. Although brute-force algorithms have their place in other contexts, they are a poor choice for the \texttt{gcd} function if you are concerned about efficiency. For example, if you call \texttt{gcd} with 1000005 and 1000000, the brute-force algorithm will run through the body of the \texttt{while} loop almost a million times before it comes up with the answer 5, even though you can instantly arrive at that result just by thinking about the two numbers.

What you need to find is an algorithm that is guaranteed to terminate with the correct answer but that requires fewer steps than the brute-force approach. This is where cleverness and a clear understanding of the problem pay off. Fortunately, the necessary creative insight was described sometime around 300 BCE by the Greek mathematician Euclid, whose \textit{Elements} (book 7, proposition II) contains an elegant solution to this problem. In modern English, Euclid’s algorithm can be described as follows:

1. Divide \texttt{x} by \texttt{y} and compute the remainder; call that remainder \texttt{r}.
2. If \texttt{r} is zero, the procedure is complete, and the answer is \texttt{y}.
3. If \texttt{r} is not zero, set \texttt{x} equal to the old value of \texttt{y}, set \texttt{y} equal to \texttt{r}, and repeat the entire process.

You can easily translate this algorithmic description into the following code:

```javascript
function gcd(x, y) {
    let r = x % y;
    while (r !== 0) {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}
```

This implementation of the \texttt{gcd} function also correctly finds the greatest common divisor of two integers. It differs from the brute-force implementation in two respects. On the one hand, it computes the result much more quickly. On the other, it is more difficult to prove correct.
Although a formal proof of correctness for Euclid’s algorithm is beyond the scope of this book, you can easily get a feel for how the algorithm works by adopting the mental model of mathematics the Greeks used. In Greek mathematics, geometry held center stage, and numbers were thought of as distances. For example, when Euclid set out to find the greatest common divisor of two whole numbers, such as 51 and 15, he framed the problem as one of finding the longest measuring stick that could be used to mark off each of the two distances involved. Thus, you can visualize the specific problem by starting out with two sticks, one 51 units long and one 15 units long, as follows:

\[
\begin{array}{c} 
\text{x} \\
\text{y} \\
\end{array}
\begin{array}{c} 
51 \\
15 \\
\end{array}
\]

The problem is to find a new measuring stick that you can lay end to end on top of each of these sticks so that it precisely covers each of the distances \(x\) and \(y\).

Euclid’s algorithm begins by marking off the large stick in units of the shorter one, like this:

\[
\begin{array}{c} 
\text{x} \\
\text{y} \\
\end{array}
\begin{array}{c} 
51 \\
15 \quad 15 \quad 15 \quad 6 \\
\end{array}
\]

Unless the smaller number is an exact divisor of the larger one, there is some remainder, as indicated by the shaded section of the lower stick. In this case, 15 goes into 51 three times with 6 left over, which means that the shaded region is 6 units long. The fundamental insight that Euclid had is that the greatest common divisor for the original two distances must also be the greatest common divisor of the length of the shorter stick and the length of the shaded region in the diagram.

Given this observation, you can solve the original problem by reducing it to a simpler problem involving smaller numbers. Here, the new numbers are 15 and 6, and you can find their greatest common divisor by reapplying Euclid’s algorithm. You start by representing the new values, \(x'\) and \(y'\), as measuring sticks of the appropriate length. You then mark off the larger stick in units of the smaller one.

\[
\begin{array}{c} 
x' \\
y' \\
\end{array}
\begin{array}{c} 
15 \\
6 \quad 6 \quad 3 \\
\end{array}
\]

Once again, this process results in a leftover region, which this time has length 3. If you then repeat the process one more time, you discover that the shaded region of length 3 is itself the common divisor of \(x'\) and \(y'\) and, therefore, by Euclid’s
Control Statements

proposition, of the original numbers \(x\) and \(y\). That 3 is indeed a common divisor of the original numbers is demonstrated by the following diagram:

\[
\begin{array}{cccccccccccc}
\times & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
y & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

Euclid supplies a complete proof of his proposition in the *Elements*. If you are intrigued by how mathematicians thought about such problems more than 2000 years ago, you may find it interesting to look up translations of the original Greek source.

Although Euclid’s algorithm and the brute-force algorithm correctly compute the greatest common divisor of two integers, there is an enormous difference in the efficiency between the two algorithmic strategies. Suppose that the two numbers are once 1000005 and 1000000. To find the greatest common divisor of these two integers, the brute-force algorithm requires a million steps; Euclid’s algorithm requires only two. At the beginning of Euclid’s algorithm, \(x\) is 1000005, \(y\) is 1000000, and \(r\) is set to 5 during the first cycle of the loop. Since the value of \(r\) is not 0, the program sets \(x\) to 1000000, sets \(y\) to 5, and starts again. On the second cycle, the new value of \(r\) is 0, so the program exits from the *while* loop and reports that the answer is 5.

The two strategies for computing greatest common divisors presented in this chapter offer a clear demonstration that the choice of algorithm can have a profound effect on the efficiency of the solution. If you continue your study of computer science, you will learn how to quantify such differences in performance along with several general approaches for improving algorithmic efficiency.

4.7 Avoiding fuzzy standards of truth

In the programs included in this book, every conditional test produces a Boolean value, which means that it will always be either *true* or *false*. Unfortunately, the JavaScript language is much less disciplined on this point. The following values (a couple of which you have not yet seen) are defined to be *falsy*, presumably to imply that they are like the legitimate Boolean value *false*:

*false*, 0, "", *undefined*, *null*, and *NaN*

Conversely, any value that is not one of these six values is said to be *truthy*. In any conditional context, any “falsy” value is treated as if it were the value *false*; any “truthy” value is treated as if it were the value *true*. 
If you want to write programs that are easy to read and maintain, you should absolutely avoid relying on these fuzzy definitions of truth and falsity and make sure—as this book does—that every test produces a legitimate Boolean value. In his book, *JavaScript: The Good Parts*, Douglas Crockford lists the “surprisingly large number of falsy values” in his appendix on JavaScript’s “awful parts.” But you might also take the following advice from an even older source:

"Let what you say be simply ‘Yes’ or ‘No’; anything more than this comes from evil."

—Matthew 5:37, *The New English Bible*

**Summary**

The purpose of this chapter is to introduce the most common control statements available in JavaScript and give you various examples of their use. The important points introduced in this chapter include:

- One of the most important types in JavaScript is *Boolean data*, for which the domain consists of just two values: `true` and `false`.
- You can generate Boolean values using the *relational operators* (`<`, `<=`, `>`, `>=`, `===`, and `!==`), and you can combine Boolean values using the *logical operators* (`&&`, `||`, and `!`).
- The logical operators `&&` and `||` are evaluated in left-to-right order in such a way that the evaluation stops as soon as the program can determine the result. This behavior is called *short-circuit evaluation*.
- Control statements fall into two classes: *conditional* and *iterative*.
- The `if` statement specifies conditional execution when a section of code should be executed only in certain cases or when the program needs to choose between two alternate paths.
- The `switch` statement specifies conditional execution when a problem has the following structure: in case 1, do this; in case 2, do that; and so forth.
- The `while` statement specifies repetition as long as some condition is met.
- The `for` statement specifies repetition in which some action is needed on each cycle in order to update the value of an index variable. The general form of the `for` statement header is
  
  ```javascript
  for (init; test; step)
  ```
  
  where `init` typically declares and initializes an index variable, `test` specifies the conditions under which the loop continues, and `step` determines what operations are performed at the end of each cycle.
• An algorithm is a strategy that is clear and unambiguous, effective, and finite.
• There are often many different algorithms for solving a particular problem. Algorithms for solving a problem often vary dramatically in their efficiency. Choosing the algorithm that best fits the application is an important part of your task as a programmer.
• JavaScript does not insist that conditional tests have the values true or false, but instead allows programmers to slip into using a fuzzier standard of truth. If you want to write programs that are easy to read and maintain, you should avoid writing any code that relies on this unfortunate feature of the language.

## Review questions

1. What are the JavaScript keywords for the two Boolean values?

2. How would you write a Boolean expression to test whether the value of the integer variable \( n \) was in the range 0 to 9, inclusive?

3. Describe in English what the following conditional expression means:

\[
(x !== 4) || (x !== 17)
\]

For what values of \( x \) is this condition true?

4. What is meant by the term short-circuit evaluation?

5. What are the two classes of control statements?

6. What does it mean to say that two control statements are nested?

7. Describe in English the general operation of the `switch` statement.

8. What rule does this chapter suggest with respect to the final statement in any `case` or `default` clause?

9. What special value is used in the `monthName` function in Figure 4-1 to indicate an illegal numeric month?

10. What was the nickname of the Small-Scale Experimental Machine developed at Manchester University that was in many respects the first modern digital computer? What computing pioneer wrote the first program for that machine?

11. Suppose the body of a `while` loop contains a statement that, when executed, causes the condition for that `while` loop to become `false`. Does the loop terminate immediately at that point or does it complete the current cycle?
12. What term do computer scientists use to refer to an incomplete program written partly in a programming language and partly in English?

13. Why is it important for the comments preceding the `digitSum` function shown in Figure 4-2 to require that the argument value be positive? What would happen if the argument were negative?

14. What is the purpose of each of the three expressions that appear in the control line of a `for` statement?

15. What `for` loop control line would you use in each of the following situations:
   a) Counting from 1 to 100.
   b) Counting by sevens starting at 0 until the number has more than two digits.
   c) Counting backward by twos from 100 to 0.

16. What conditions must a solution strategy meet in order to be an algorithm?

17. Use Euclid’s algorithm to compute the greatest common divisor of 7735 and 4185. What values does the local variable `r` take on during the calculation?

18. In the examples of Euclid’s algorithm to calculate `gcd(x, y)` that appear in this chapter, `x` is always larger than `y`. What happens if `x` is smaller than `y`?

19. What do the terms `falsy` and `truthy` signify in JavaScript? What strategy does this book suggest for avoiding the ambiguity associated with these terms?

---

**Exercises**

1. Using the two definitions of the `max` function—one using an `if` statement and one using the `?:` operator—as examples, write corresponding implementations of the `min` function, which returns the smaller of its two arguments.

2. Write a function `max3` that returns the largest of its three arguments.

3. As a way to pass the time on long bus trips, young people growing up in the United States have been known to sing the following rather repetitive song:

   99 bottles of beer on the wall.
   99 bottles of beer.
   You take one down, pass it around.
   98 bottles of beer on the wall.

   98 bottles of beer on the wall. . .

   Anyway, you get the idea. Write a JavaScript program to display the lyrics of this song using `console.log`. In testing your program, it would make sense to use some constant other than 99 as the initial number of bottles.
4. While we’re on the subject of silly songs, another old standby is “This Old Man,” for which the first verse is

This old man, he played 1.
He played knick-knack on my thumb.
With a knick-knack, paddy-whack,
Give your dog a bone.
This old man came rolling home.

Each subsequent verse is the same, except for the number and the rhyming word at the end of the second line, which get replaced as follows:

2—shoe 5—hive 8—pate
3—knee 6—sticks 9—spine
4—door 7—heaven 10—shin

Write a program to display all 10 verses of this song.

5. Write a function that takes a positive integer \( N \) and then calculates and displays the sum of the first \( N \) odd integers. For example, if \( N \) is 4, your function should display the value 16, which is 1 + 3 + 5 + 7.

6. Why is everything either at sixes or at sevens?
—Gilbert and Sullivan, H.M.S. Pinafore, 1878

Write a program that displays the integers between 1 and 100 that are divisible by either 6 or 7 but not both.

7. Using the digitSum function as a model, define a function that takes a number and returns the number containing the same digits in the reverse order, as illustrated by the following sample run:

```
JavaScript Console
> reverseDigits(1729)
9271
> reverseDigits(123456789)
987654321
> 
```

The idea in this exercise is not to take the integer apart character by character, which you will not learn how to do until Chapter 7. Instead, you need to use arithmetic to compute the reversed integer as you go. For example, in the call to `reverseDigits(1729)`, the new integer will be 9 after the first cycle of the loop, 92 after the second, 927 after the third, and 9271 after the fourth.

8. The digital root of an integer \( n \) is defined as the result of summing the digits repeatedly until only a single digit remains. For example, the digital root of 1729 can be calculated using the following steps:
Because the total at the end of step 3 is the single digit 1, that value is the digital root. Write a function `digitalRoot` that returns this value.

9. Rewrite the `Countdown` program given in Figure 4-4 so that it uses a `while` loop instead of a `for` loop.

10. Write a function `drawConsoleBox(width, height)` that draws a box on the console with the specified dimensions. The corners of the box should be represented using a plus sign (+), the top and bottom borders using a minus sign (-), and the left and right borders using a vertical bar (|). For example, calling `drawConsoleBox(52, 6)` should produce the following diagram:

```
+-----------------------------------+
|                                  |
|                                  |
|                                  |
+-----------------------------------+
```

11. Write a function `drawConsoleArrow(width)` that draws a triangular arrow pointing to the right in which the center line has the specified width. For example, calling `drawConsoleArrow(7)` should create the following output:

```
*  
** 
***
****
*****
******
*******
```

This program is easy to write if you simply add a second loop that counts backwards at the end of the code from `drawConsoleTriangle`. You can get a better sense of the flexibility of the `for` statement in JavaScript if you instead use a single outer loop that changes direction when it reaches the desired width.

12. Write a function `drawConsolePyramid(height)` that draws a pyramid of the specified height in which the width of each row increases by two as you move downward on the console. Each of the rows should be centered with
13. Write a program that displays a pyramid on the graphics window. The pyramid consists of bricks arranged in horizontal rows, arranged so that the number of bricks in each row decreases by one as you move upward, as follows:

The pyramid should be centered in the window both horizontally and vertically and should use constants to define the dimensions of each brick and the height of the pyramid.

14. Enhance the `Checkerboard.js` program shown in Figure 4-5 so that the graphics window also displays the red and black checkers corresponding to the initial state of the game, which looks like this:
The other change in this program is that the color of the dark squares has been changed from black to gray so that the black checkers are not lost against the background.

15. In mathematics, there is a famous sequence of numbers called the **Fibonacci sequence** after the thirteenth-century Italian mathematician Leonardo Fibonacci. The first two terms in this sequence are 0 and 1, and every subsequent term is the sum of the preceding two. Thus the first several terms in the Fibonacci sequence are as follows:

\[
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_2 &= 1 \ (0 + 1) \\
F_3 &= 2 \ (1 + 1) \\
F_4 &= 3 \ (1 + 2) \\
F_5 &= 5 \ (2 + 3) \\
F_6 &= 8 \ (3 + 5)
\end{align*}
\]

Write a program to display the values in this sequence from \(F_0\) through \(F_{15}\).

16. Modify the program in the preceding exercise so that instead of specifying the index of the final term, the program displays those terms in the Fibonacci sequence that are smaller than 10,000.

17. An integer greater than 1 is said to be **prime** if it has no divisors other than itself and one. The number 17, for example, is prime, because it has no factors other than 1 and 17. The number 91, however, is not prime because it is divisible by 7 and 13. Write a predicate function \(\text{isPrime}(n)\) that returns \(\text{true}\) if the integer \(n\) is prime, and \(\text{false}\) otherwise. As an initial strategy, implement \(\text{isPrime}\) using a brute-force algorithm that simply tests every possible divisor. Once you have that version working, try to come up with improvements to your algorithm that increase its efficiency without sacrificing its correctness.

18. Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of \(n\) is any divisor less than \(n\) itself). They called such numbers **perfect numbers**. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.

Write a predicate function \(\text{isPerfect}(n)\) that returns \(\text{true}\) if the integer \(n\) is perfect, and \(\text{false}\) otherwise. Test your implementation by writing a program that uses the \(\text{isPerfect}\) function to check for perfect numbers in the range 1 to 9999 by testing each number in turn. Whenever it identifies a
perfect number, your program should display that number on the screen. The first two lines of output should be 6 and 28. Your program should find two other perfect numbers in that range as well.

19. Although Euclid’s algorithm for calculating the greatest common divisor is one of the oldest to be dignified with that term, there are other algorithms that date back many centuries. In the Middle Ages, one of the problems that required sophisticated algorithmic thinking was determining the date of Easter, which falls on the first Sunday after the first full moon following the vernal equinox. Given this definition, the calculation involves interacting cycles of the day of the week, the orbit of the moon, and the passage of the sun through the zodiac. Early algorithms for solving this problem date back to the third century and are described in the writings of the eighth-century scholar known as the Venerable Bede. In 1800, the German mathematician Carl Friedrich Gauss published an algorithm for determining the date of Easter that was purely computational in the sense that it relied on arithmetic rather than looking up values in tables. His algorithm—translated from the German—appears in Figure 4-9.

Write a JavaScript function `findEaster(year)` that returns a string showing the date of Easter in the specified year. For example, calling `findEaster(1800)` returns the string “April 13” because that is the date of Easter in the year that Gauss published his algorithm.

Unfortunately, the algorithm in Figure 4-9 works only for years in the 18th and 19th centuries. It is easy, however, to search the web for extensions that work for all years. Once you have completed your implementation of Gauss’s algorithm, undertake the necessary research to implement a more general approach.

---

**FIGURE 4-9** Gauss’s algorithm for computing the date of Easter

I. Divide the number of the year for which one wishes to calculate Easter by 19, by 4, and by 7, and call the remainders of these divisions a, b, and c, respectively. If the division is even, set the remainder to 0; the quotients are not taken into account. Precisely the same is true of the following divisions.

II. Divide the value $19a + 23$ by 30 and call the remainder d.

III. Finally, divide $2b + 4c + 6d + 3$, or $2b + 4c + 6d + 4$, choosing the former for years between 1700 and 1799 and the latter for years between 1800 and 1899, by 7 and call the remainder e. Then Easter falls on March $22 + d + e$, or when $d + e$ is greater than 9, on April $d + e - 9$.

Translated from Karl Friedrich Gauss, “Berechnung des Osterfestes,” August 1800
http://gdz.sub.uni-goettingen.de/no_cache/dms/load/img/?IDDOC=137484
Our module structure is based on the decomposition criteria known as information hiding. According to this principle, system details that are likely to change independently should be the secrets of separate modules.

— David Parnas, Paul Clements, and David Weiss,
“The modular structure of complex systems,” 1984

David Parnas is Professor of Software Engineering emeritus at the University of Limerick in Ireland, where he directed the Software Quality Research Laboratory, and has also taught at universities in Germany, Canada, and the United States. His most influential contribution to software engineering is his groundbreaking 1972 paper entitled “On the criteria to be used in decomposing systems into modules,” which provided much of the foundation for the strategy of decomposition described in this chapter. Professor Parnas also attracted considerable public attention in 1985 when he resigned from a Department of Defense panel investigating the software requirements of the proposed Strategic Defense Initiative—more commonly known as “Star Wars”—on the grounds that the requirements of the system were impossible to achieve. For his courageous stand in bringing these problems to light, Parnas received the 1987 Norbert Wiener Award from Computer Professionals for Social Responsibility.
This chapter examines in more detail the concept of a function, which was initially presented in Chapter 1 and then revisited in the context of JavaScript in Chapter 2. A function is a set of statements that have been collected together and given a name. Because functions allow the programmer to invoke the entire set of operations using a single name, programs become much shorter and much simpler. Without functions, programs would become unmanageable as they increased in size and sophistication.

In order to appreciate how functions reduce the complexity of programs, you need to understand the concept using two distinct philosophical approaches, *reductionism* and *holism*. *Reductionism* is the philosophical principle that the whole of an object can best be understood by understanding the parts that make it up. Its antithesis is *holism*, which recognizes that the whole is often more than the sum of its parts. As you try to master the discipline of dividing large programs into functions, you must learn to see the process from each of these perspectives. If you concentrate only on the big picture, you will end up not understanding the tools you need for solving problems. However, if you focus exclusively on details, you will invariably miss the forest for the trees.

When you are first learning about programming, the best approach is usually to alternate between these two perspectives. Taking the holistic view helps sharpen your intuition about the programming process and enables you to stand back from a program and say, “I understand what this function does.” Taking the reductionistic view allows you to say, “I understand how this function works.” Both perspectives are essential. You need to understand how functions work so that you can code them correctly. At the same time, you must be able to take a step backward and look at functions holistically, so that you also understand why they are important and how to use them effectively.

### 5.1 A quick review of functions

You have been working with functions ever since you wrote your first Karel program in Chapter 1, but you have so far seen only a part of the computational power that functions provide. Before delving more deeply into the details of how functions work, it helps to review some basic terminology. First of all, a function consists of a set of statements that have been collected together and given a name. The act of executing the set of statements associated with a function is known as calling that function. To indicate a function call in JavaScript, you write the name of the function, followed by a list of expressions enclosed in parentheses. These expressions, which are called arguments, allow the caller to pass information to the function.
The syntax of a function definition

A typical function definition has the form shown in the syntax box on the right. The name component of this pattern indicates the function name, parameters is the list of parameter names that receive the values of the arguments, and statements represents the body of the function. Functions that return a value to the caller must contain at least one return statement that specifies the value of the function, as illustrated in the second syntax box.

These syntactic patterns are illustrated in the definition of the max function from Chapter 4, which looks like this:

```javascript
function max(x, y) {
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

This function has the name max and takes two parameters, x and y. The statements in the body decide which of these two values is larger and then return that value.

Functions, however, are often called simply for their effect and need not return an explicit value. For example, the Karel functions in Chapter 1 and the functions in Chapter 3 that implement complete programs don’t include a return statement. Some languages distinguish a function that returns a value from one that doesn’t by calling the latter a procedure. JavaScript uses the term function for both types. This terminology is technically accurate because JavaScript functions always return a value, which is the special value undefined if no return statement appears.

Parameter passing

The most important thing to remember about the process of calling a function is that the values of the arguments are copied to the parameter variables in the order in which they appear. The first argument is assigned to the first parameter variable, the second argument to the second parameter variable, and so on. The names of the parameters have nothing to do with the order in which their values are assigned. There may well be a variable named x in both the calling function and in the parameter list for the function being called. That reuse of the same name, however, is merely a coincidence. Local variable names and parameter names are visible only inside the function in which their declarations appear.

Unlike most modern languages, JavaScript does not check whether the number of arguments in the call matches the number of parameters specified for the
function. If there are too many arguments, the extra ones are not assigned to any of the parameters and are therefore essentially ignored. If there are too few, any parameters that don’t have a corresponding argument are set to undefined.

JavaScript’s interpretation of missing arguments makes it easy to write functions that take optional parameters. For example, the following function displays \( n \) consecutive integers, starting with the value \( \text{start} \) if a second argument is supplied and with the value 1 if that argument is missing:

```javascript
function count(n, start) {
  if (start === undefined) start = 1;
  for (let i = 0; i < n; i++) {
    console.log(start + i);
  }
}
```

The following console session illustrates the operation of `count`, both when it is given a second argument and when it is not:

```
> count(3);
1
2
3
> count(2, 10);
10
11
> 
```

As written, the first line of this function is entirely self-explanatory. If the value of \( \text{start} \) is undefined, set it to 1. Unfortunately, setting a default value is a situation in which JavaScript programmers often try to be too clever. Instead of checking explicitly to see whether `count` is equal to `undefined`, they rely on JavaScript’s fuzzy definitions of truth and falsity and write either

```javascript
if (!start) start = 1;
```

or, depending on an interpretation of the JavaScript `||` operator that it is far safer not to know, the even more cryptic

```javascript
start = start || 1;
```

Both of these overly abbreviated forms depend on the fact that JavaScript interprets `undefined` as false. By not checking for `undefined` explicitly, however, these attempts to shorten the code actually make it buggy by making it impossible to
count starting from 0, which is also interpreted as false. Thus, if you were to use either of these misconceived shorthand forms, calling `count(10, 0)` would seemingly ignore your clear intention of starting with 0 and count from 1 to 10.

**Predicate functions**

As you have seen in the earlier chapters, functions can return values of different data types. For example, the function `createFilledCircle` on page 88 returns a `GOval`, the `monthName` on page 114 returns a string, and the function `fact` on page 120 returns a number. Although functions in JavaScript can return values of any type, functions that return Boolean values deserve special attention because they play such an important role in programming. Functions that return Boolean values are called *predicate functions*.

As noted earlier in the chapter, there are only two Boolean values: `true` and `false`. Thus a predicate function—no matter how many arguments it takes or how complicated its internal processing may be—must eventually return one of these two values. The process of calling a predicate function is therefore analogous to asking a yes/no question and getting an answer.

As an example, the following function definition answers the question “is `n` an even number?” for a particular integer `n` supplied by the caller as an argument:

```javascript
function isEven(n) {
    return n % 2 === 0;
}
```

A number is even if there is no remainder when that number is divided by two. If `n` is even, the expression

```
n % 2 === 0
```

therefore has the value `true`, which is returned as the result of the `isEven` function. If `n` is odd, the function returns `false`. Because `isEven` returns a Boolean result, you can use it directly in a conditional context. For example, the following `for` loop uses `isEven` to display all the even numbers between 1 and 100, inclusive:

```javascript
for (let i = 1; i <= 100; i++) {
    if (isEven(i)) {
        console.log(i);
    }
}
```

The `for` loop runs through each number, and the `if` statement asks the simple question “is this number even?” If it is, the program calls `console.log` to display the number; if not, nothing happens.
If you are writing a program that works with dates, you may discover that you would like to write a predicate function \textit{isLeapYear} that determines whether a given year qualifies as a leap year. Although we tend to think of leap years as occurring once every four years, astronomical realities are not quite so tidy. Because it takes about a quarter of a day more than 365 days for the earth to complete its annual orbital circuit around the sun, adding an extra day once every four years does help to keep the calendar in sync with the sun, but it is still off by a slight amount. To ensure that the beginning of the year does not slowly drift through the seasons, the rule used for leap years is in fact more complicated. Leap years come every four years, except for years ending in 00, which are leap years only if they are divisible by 400. Thus, 1900 was not a leap year even though 1900 is divisible by 4. The year 2000, on the other hand, was a leap year because 2000 is divisible by 400. For any leap year, one of the following conditions must hold:

- The year is divisible by 4 but not divisible by 100, \textit{or}
- The year is divisible by 400.

Although this rule is more complicated than the one-year-in-every-four approach that works most of the time, it is easy to code the correct rule in JavaScript as a predicate function, as follows:

\begin{verbatim}
function isLeapYear(int year) {
  return ((year % 4 === 0) && (year % 100 !== 0)) || (year % 400 === 0);
}
\end{verbatim}

Once the function is defined, you can test for leap years like this:

\begin{verbatim}
if (isLeapYear(year)) . . .
\end{verbatim}

5.2 Libraries

Writing a program to solve a large or difficult problem forces you to manage a large amount of complexity. There are algorithms to design, special cases to consider, user requirements to meet, and innumerable details to get right. To make programming manageable, you must reduce the complexity of the programming process as much as possible. Functions reduce some of the complexity; libraries offer a similar reduction in programming complexity but at a higher level of detail. A function gives its caller access to a set of steps that implements a single operation. A library provides access to a set of definitions that implements what computer scientists describe as a \textit{programming abstraction}. The extent to which a particular abstraction simplifies the programming process, however, depends on how well you have designed it.
Creating your own libraries

You can define a JavaScript library simply by combining the relevant definitions into a file whose name ends with the standard .js extension. For example, if you work extensively with dates in your programming, you could combine the constant definitions for the month names, the \texttt{monthName} function from page 114, and the \texttt{isLeapYear} function from page 148 into a library called \texttt{DateLib.js} as shown in Figure 5-1 on the next page.

Once you have created the \texttt{DateLib.js} library, you can then use it in much the same way that you use any other library. All you need to do is add the line

\[ \text{<script type="text/javascript" src="DateLib.js"></script>} \]

to the \texttt{index.html} file. Your JavaScript program will then have access to the constants for the month names and the functions \texttt{monthName} and \texttt{isLeapYear}. In computer science terminology, the \texttt{DateLib.js} library \textit{exports} these constants and functions, which are collectively called \textit{entries}.

The principle of information hiding

One of the goals of any library is to hide the complexity involved in the underlying implementation. By exporting the \texttt{isLeapYear} function, the \texttt{DateLib.js} library hides away the complexities involved in determining whether years ending in 00 are leap years. When you call the \texttt{isLeapYear} function, you don’t need to have any idea how the implementation works. In fact, you don’t even have to know that the special rules for century years exist. Those details are relevant only to the programmers responsible for implementing the \texttt{DateLib.js} library.

Knowing how to call the \texttt{isLeapYear} function and knowing how to implement it are both important skills. It is important to recognize, however, that those two skills—calling a function and implementing one—are to a large extent independent. Successful programmers often use functions that they wouldn’t have a clue how to write. Conversely, programmers who implement a library function can never anticipate all the potential uses for that function.

To emphasize the difference in perspective between programmers who implement a library and those who use it, computer scientists have assigned names to programmers working in each of these roles. Naturally enough, a programmer who implements a library is called an \textit{implementer}. Conversely, a programmer who calls functions provided by a library is called a \textit{client} of that library. As you go through the chapters in this book, you will have a chance to look at several libraries from both perspectives, first as a client and later as an implementer.
A simple library for working with dates

/*
 * File: DateLib.js
 * 
 * The library exports the functions monthName and isLeapYear, along with
 * a set of constants giving names to the months of the year. It is not
 * intended as a complete set of facilities for dates but instead serves
 * as an example of a JavaScript library.
 */

/* Constants for the names of the months */

const JANUARY = 1;
const FEBRUARY = 2;
const MARCH = 3;
const APRIL = 4;
const MAY = 5;
const JUNE = 6;
const JULY = 7;
const AUGUST = 8;
const SEPTEMBER = 9;
const OCTOBER = 10;
const NOVEMBER = 11;
const DECEMBER = 12;

/*
 * Converts a numeric month in the range 1 to 12 into its name.
 */

function monthName(month) {
    switch (month) {
    case JANUARY: return "January";
    case FEBRUARY: return "February";
    case MARCH: return "March";
    case APRIL: return "April";
    case MAY: return "May";
    case JUNE: return "June";
    case JULY: return "July";
    case AUGUST: return "August";
    case SEPTEMBER: return "September";
    case OCTOBER: return "October";
    case NOVEMBER: return "November";
    case DECEMBER: return "December";
    default: return undefined;
    }
}

/*
 * Returns true if the specified year is a leap year and false otherwise.
 */

function isLeapYear(year) {
    return year % 400 === 0 || (year % 4 === 0 && year % 100 !== 0);
}
Both functions and libraries offer a tool for hiding lower-level implementation details so that clients need not worry about them. In computer science, this technique is called information hiding. The fundamental idea, which was championed by David Parnas in the early 1970s, is that the complexity of programming systems is best managed by making sure that details are visible only at those levels of the program at which they are relevant. For example, only the programmers who implement isLeapYear need to know the details of its operation. Clients who merely use isLeapYear can remain blissfully unaware of the underlying details.

Creating a library to support randomness

To give you more insight into how libraries work, the next few subsections define a library called RandomLib.js that exports a few useful tools that allow you to write programs that make seemingly random choices. Being able to simulate random behavior is necessary, for example, if you want to write a computer game that involves flipping a coin or rolling a die, but is also useful in more practical contexts. Programs that simulate random processes are said to be nondeterministic.

The starting point for the RandomLib.js library is the random function, which was listed in Figure 2-3 as one of the functions in the Math class. Calling Math.random returns a number in the range beginning at 0 and extending up to but not including the value 1. Every now and then, there is an application for which you need random values in precisely this range. More often than not, however, you would like to generate random values in a more general way. For example, if you are working on a game program that involves rolling a die, you would like to be able to produce a random integer between 1 and 6, inclusive. To simulate flipping a coin, it would be useful to have a function that produced a Boolean value so that true and false occurred with equal probability.

The first and probably most important step in designing the RandomLib.js library consists of choosing what functions it should export. Those functions should be simple to use and should hide as much of the underlying complexity as possible. They should also provide the functionality necessary to meet the needs of a wide range of clients, which means that you need to have some idea of what operations clients are likely to need. Understanding those needs depends in part on your own experience, but often requires interacting with potential clients to get a better sense of their requirements.

From my own experience with programming, I know that the operations clients expect from a random-number library include the following:
- **Selecting a random integer in a specified range.** If you want, for example, to simulate the process of rolling a standard six-sided die, you need to choose a random integer between 1 and 6.

- **Choosing a random real number in a specified range.** If you want to position an object at a random point in space, you need to choose random $x$ and $y$ coordinates within whatever limits are appropriate to the application.

- **Simulating a random event with a specific probability.** If you want to simulate flipping a coin, you need to generate the value heads with probability 0.5, which corresponds to 50 percent of the time.

- **Picking a random color.** In certain graphical applications, it is useful to choose a color at random to create unexpected patterns on the screen.

Translating these conceptual operations into a set of functions is a reasonably straightforward task, especially if you look at the problem from the client’s perspective. The four functions exported by `RandomLib.js` are `randomInteger`, `randomReal`, `randomChance`, and `randomColor`, which correspond directly to the four operations clients expected to use. The complete code for `RandomLib.js` appears in Figure 5-2, along with comments designed to help the client use these functions.

Although the client’s view of these functions is relatively easy to understand, the implementations of these functions all involve some level of complexity that should be hidden from the client. The rest of this section walks through the code for each of these functions in detail.

As you can see from the comments in Figure 5-2, the `randomInteger` function takes two integers and returns an integer in the inclusive range that extends from the first argument to the second. If you wanted to simulate a die roll, you would call

```
randomInteger(1, 6)
```

To simulate a spin on a European roulette wheel (American wheels have both a 0 and a 00 slot in addition to the numbers from 1 to 36), you would call

```
randomInteger(0, 36)
```

The body of the `randomInteger` function fits on a single line, but it takes some thought to understand what that line does. For any supplied values of the parameters `low` and `high`, the `randomInteger` function returns the value of the following expression:

```
low + Math.floor((high - low + 1) * Math.random())
```
```javascript
 Figure 5-2 A simple random library

/*
 * File: RandomLib.js
 * -------------------
 * This file contains a simple library of functions to generate random
 * integers, reals, booleans, and colors.
 */

/*
 * Returns a random integer in the range low to high, inclusive.
 */
function randomInteger(low, high) {
    return Math.floor((high - low + 1) * Math.random());
}

/*
 * Returns a random real number in the half-open interval [low, high).
 */
function randomReal(low, high) {
    return low + (high - low) * Math.random();
}

/*
 * Returns true with probability p. If p is missing, it defaults to 0.5.
 */
function randomChance(p) {
    if (p === undefined) p = 0.5;
    return Math.random() < p;
}

/*
 * Returns a random opaque color expressed as a string consisting of a "#"
 * followed by six random hexadecimal digits.
 */
function randomColor() {
    let str = "#";
    for (let i = 0; i < 6; i++) {
        switch (randomInteger(0, 15)) {
            case 0: case 1: case 2: case 3: case 4:
                case 5: case 6: case 7: case 8: case 9: str += i; break;
            case 10: str += "A"; break;
            case 11: str += "B"; break;
            case 12: str += "C"; break;
            case 13: str += "D"; break;
            case 14: str += "E"; break;
            case 15: str += "F"; break;
        }
    }
    return str;
}
```
It is probably easiest to examine this expression from the inside out. The function call to `Math.random` returns a random number that can be as small as 0 but is always strictly less than 1. In mathematics, a range of real numbers that can be equal to one endpoint but not the other is called a half-open interval. On a number line, a half-open interval is marked using an open circle to show that the endpoint is excluded, like this:

\[
\begin{align*}
0 & \quad \quad \quad \quad 1 \\
\end{align*}
\]

In mathematics, the standard convention is to use square brackets to indicate closed ends of intervals and parentheses to indicate open ones, so that the notation \([0, 1)\) indicates the half-open interval corresponding to this diagram.

The next step is to multiply the random value in the \([0, 1)\) interval by the number of possible outcomes, which is given by the expression

\[
(high - low + 1)
\]

Having to add one after subtracting `low` from `high` might at first seem confusing, but this situation is precisely analogous to the fencepost problem introduced in Chapter 1. Subtracting `low` from `high` gives the distance between these points, which therefore corresponds to the length of the fence. The number of possible outcomes is the number of integers covered by the range, keeping in mind that defining the range to be inclusive means that there is an integer—corresponding to a fencepost—at both ends. In the die roll example, the length of the range is 5 (6 – 1), but the number of possible outcomes is 6. Multiplying the result of `Math.random` by 6 produces a real number in the \([0, 6)\) range, as follows:

\[
\begin{align*}
0 & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \\
\end{align*}
\]

The code for `randomInteger` then uses `Math.floor` to convert the real number to an integer by rounding it down to the next smallest whole number. From there, the last remaining step is to add the value of `low` so that the set of possible return values for `randomInteger` starts at the correct point, as illustrated on the following number line in which only the solid dots represent possible values:

\[
\begin{align*}
1 & \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \\
\end{align*}
\]

The implementation of `randomReal` follows much the same strategy as the code for `randomInteger` but is simpler because it can leave out both the call to `Math.floor` and the adjustment of the range to avoid the fencepost problem, which does not apply when real numbers are involved. The code is therefore simply
function randomReal(low, high) {
    return low + (high - low) * Math.random();
}

The function randomChance is used to simulate random events that occur with some fixed probability. In accord with mathematical convention, a probability is represented as a number between 0 and 1, where 0 means that the event never occurs and 1 means that it always does. Calling randomChance(p) returns true with probability p. Thus, calling randomChance(0.75) returns true 75 percent of the time. The only additional aspect of the code is that the implementation checks to see whether a value of p was supplied and, if not, substitutes the value 0.5 to indicate a 50-50 chance.

You can use randomChance to simulate flipping a coin, as illustrated by the following function, which returns "heads" or "tails" with equal probability:

function flipCoin() {
    if (randomChance()) {
        return "heads";
    } else {
        return "tails";
    }
}

The only remaining function in the RandomLib.js library is randomColor, which returns a random color value among the 16,777,216 possible colors available in JavaScript, not counting those that include some level of transparency. As described on page 75, JavaScript allows you to specify any of these colors using the standard web convention of writing a hashtag symbol (#) followed by six digits in hexadecimal, or base 16. Hexadecimal notation is discussed in detail in Chapter 7, but all you need to know to understand the implementation of randomColor is that hexadecimal notation augments the familiar digits 0 through 9 with the letters from A to F to account for the six extra digits required to represent a number in base 16.

The code for randomColor in Figure 5-2 initializes the variable str to the string "#" and then concatenates six additional characters to the end, randomly chosen from the 16 possible hexadecimal digits.

**Using the random library**

As an illustration of how clients might use the RandomLib.js library, the Craps function in Figure 5-3 uses the randomInteger function to play the casino game called craps. The rules for craps appear in the comments at the beginning of the program, although you would probably want the program to explain the rules to the user as well. In this example, the printed instructions have been omitted to ensure
* File: Craps.js

This program plays the casino game of Craps. At the beginning of the game, the player rolls a pair of dice and computes the total. If the total is 2, 3, or 12 (called "craps"), the player loses. If the total is 7 or 11 (called a "natural"), the player wins. If the total is any other number, that number becomes the "point." From here, the player keeps rolling the dice until (a) the point comes up again, in which case the player wins or (b) a 7 appears, in which case the player loses. The numbers 2, 3, 11, and 12 no longer have special significance after the first roll.

function Craps() {
    let total = rollTwoDice();
    if (total === 7 || total === 11) {
        console.log("That's a natural. You win.");
    } else if (total === 2 || total === 3 || total === 12) {
        console.log("That's craps. You lose.");
    } else {
        let point = total;
        console.log("Your point is " + point + ".");
        let running = true;
        while (running) {
            total = rollTwoDice();
            if (total === point) {
                console.log("You made your point. You win.");
                running = false;
            } else if (total === 7) {
                console.log("That's a 7. You lose.");
                running = false;
            }
        }
    }
}

function rollTwoDice() {
    let d1 = randomInteger(1, 6);
    let d2 = randomInteger(1, 6);
    let total = d1 + d2;
    console.log("Rolling dice: " + d1 + " + " + d2 + " = " + total);
    return total;
}
that the code fits on a single page. The code itself essentially follows the outline imposed by the rules of the game. In particular, it rolls the dice initially and then chooses how to proceed based on the result of that first roll. Moreover, because the task of rolling two dice and determining their sum appears at two different points in the program, it makes sense to decompose that part of the program into a separate function.

Although the Craps function is nondeterministic and will therefore produce different outcomes each time, one possible console session looks like this:

```
> Craps();
Rolling dice: 2 + 5 = 7
That's a natural. You win.
> Craps();
Rolling dice: 4 + 5 = 9
Your point is 9.
Rolling dice: 6 + 2 = 8
Rolling dice: 5 + 6 = 11
Rolling dice: 3 + 4 = 7
That's a 7. You lose.
> Craps();
Rolling dice: 6 + 2 = 8
Your point is 8.
Rolling dice: 2 + 6 = 8
You made your point. You win.
> Craps();
Rolling dice: 2 + 1 = 3
That's craps. You lose.
```

As a second example of a program that uses the RandomLib.js library, the RandomCircles.js program in Figure 5-4 at the top of the next page displays circles of various random sizes, random colors, and random positions, subject to the restriction that the entire circle must always fit inside the graphics window. The display will be different each time, but one possible run of the RandomCircles.js program looks like this:
Functions

Although you can certainly get by with an intuitive understanding of how the function-calling process works, it sometimes helps to understand precisely what happens when one function calls another in JavaScript. The sections that follow describe the process in detail and then walk you through a simple example designed to help you visualize exactly what is going on.

5.3 The mechanics of function calls

/*
 * File: RandomCircles.js
 * ---------------------
 * This program draws a set of 10 circles with different sizes, positions,
 * and colors. Each circle has a randomly chosen color, a randomly chosen
 * radius within a specified range, and a randomly chosen position subject
 * to the condition that the circle must fit inside the graphics window.
 */

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const N_CIRCLES = 10;
const MIN_RADIUS = 15;
const MAX_RADIUS = 50;

/* Main program */

function RandomCircles() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    for (let i = 0; i < N_CIRCLES; i++) {
        gw.add(createRandomCircle());
    }
}

/* Creates a randomly generated circle. The radius is chosen randomly
 * between MIN_RADIUS and MAX_RADIUS, the location is chosen so that the
 * circle fits in the window, and the circle is given a random color.
 */

function createRandomCircle() {
    let r = randomReal(MIN_RADIUS, MAX_RADIUS);
    let x = randomReal(r, GWINDOW_WIDTH - r);
    let y = randomReal(r, GWINDOW_HEIGHT - r);
    let circle = G Oval(x - r, y - r, 2 * r, 2 * r);
    circle.setFill(true);
    circle.setColor(randomColor());
    return circle;
}
The steps in calling a function

Whenever a function call occurs, the JavaScript interpreter executes the following operations:

1. The calling function computes values for each argument using the bindings of local variables in its own context. Because the arguments are expressions, this computation can involve operators and other functions; the calling function evaluates these expressions before execution of the new function begins.

2. The system creates new space for all the local variables required by the new function, including any parameters. These variables are allocated together in a block, which is called a stack frame.

3. The value of each argument is copied into the corresponding parameter variable. For functions with more than one argument, these copies occur in order; the first argument is copied into the first parameter, and so forth. If there are more arguments than parameters, the extra argument values play no role in the initialization of the parameters. If there are more parameters than arguments, those parameters that don’t have a corresponding argument are initialized to the value undefined.

4. The statements in the function body are executed until the program encounters a return statement or there are no more statements to execute.

5. The value of the return expression, if any, is evaluated and returned as the value of the function.

6. The stack frame created for this function call is discarded. In the process, all local variables disappear.

7. The calling program continues, with the returned value substituted in place of the call. The point to which the function returns is called the return address.

Although this process may seem to make at least some sense, you probably need to work through an example or two before you understand it fully. Reading through the example in the next section will give you some insight into the process, but it is probably even more helpful to take one of your own programs and walk through it at the same level of detail. And while you can trace through a program on paper or a whiteboard, it may be better to get yourself a supply of 3×5 index cards and then use a card to represent each stack frame. The advantage of the index-card model is that you can create a stack of index cards that closely models the operation of the computer. Calling a function adds a card; returning from the function removes it.

The combinations function

The function-calling process is most easily illustrated in the context of a specific example. Suppose that you have a collection of six coins, which in the United
States might be a penny, a nickel, a dime, a quarter, a half-dollar, and a dollar. Given those six coins, how many ways are there to choose two of them? As you can see from the full enumeration of the possibilities in Figure 5-5, the answer is 15. As a computer scientist, you should immediately think about the more general question: given a set containing $n$ distinct elements, how many ways can you choose a subset with $k$ elements? The answer to that question is computed by the \textit{combinations function} $C(n, k)$, which is defined as follows:

$$C(n, k) = \frac{n!}{k! \times (n-k)!}$$

where the exclamation point indicates the factorial function, which you saw in Chapter 4. The code to compute the combinations function in JavaScript appears in Figure 5-6.

\textbf{FIGURE 5-5} Illustration of the combinations function

\textit{If you start with six coins}

\begin{itemize}
  \item Penny + Nickel
  \item Penny + Dime
  \item Penny + Quarter
  \item Penny + Half-dollar
  \item Penny + Dollar
  \item Nickel + Dime
  \item Nickel + Quarter
  \item Nickel + Half-dollar
  \item Nickel + Dollar
  \item Dime + Quarter
  \item Dime + Half-dollar
  \item Dime + Dollar
  \item Quarter + Half-dollar
  \item Quarter + Dollar
  \item Half-dollar + Dollar
\end{itemize}

\textit{There are 15 ways to choose two coins:}
As you can see from Figure 5-6, the Combinations.js file contains two functions. The combinations function computes the value of \( C(n, k) \), and the now-familiar fact function computes factorials. A console session showing just one call to the combinations function might look like this:

```
> combinations(6, 2)
15
```

**Tracing the combinations function**

While the combinations function is interesting in its own right, the purpose of this example is to illustrate the steps involved in calling functions. When the user enters a function call in the console window, the JavaScript interpreter invokes the standard steps in the function-calling process.

As always, the first step is to evaluate the arguments in the current context. In the current example, the arguments are the numbers 6 and 2, so the evaluation...
process simply keeps track of these two values. The second step is to create a frame for the `combinations` function that contains space for the variables that are stored as part of that frame, which are the parameters and any variables that appear in declarations within the function. The `combinations` function has two parameters and no local variables, so the frame only requires enough space for the parameter variables `n` and `k`. After the frame is created, the JavaScript interpreter copies the argument values into these variables in order. Thus, the parameter variable `n` is initialized to 6, and the parameter variable `k` is initialized to 2.

In the diagrams in this book, each stack frame appears as a rectangle surrounded by a double line. Each stack-frame diagram shows the code for the function along with a pointing-hand icon that makes it easy to keep track of the current execution point. The frame also contains labeled boxes for each of the local variables. The stack frame for the `combinations` function therefore looks like this after the parameters are initialized but before execution of the function begins:

```
function combinations(n, k) {
  return fact(n) / (fact(k) * fact(n - k));
}
```

To compute the value of the `combinations` function, the program must make three calls to the function `fact`. In JavaScript, those function calls can happen in any order, but it’s easiest to process them from left to right. The first call, therefore, is the call to `fact(n)`, which is shown inside a box in the following diagram:

```
function combinations(n, k) {
  return fact(n) / (fact(k) * fact(n - k));
}
```

To evaluate this function, the system must create yet another stack frame, this time for the function `fact` with an argument value of 6. The frame for `fact` has both parameters and local variables. The parameter `n` is initialized to the value of the calling argument and therefore has the value 6. The two local variables, `i` and `result`, have not yet been initialized and therefore contain the value `undefined`, which is indicated in stack diagrams as an empty box. The new frame for `fact` gets stacked on top of the old one, which allows the JavaScript interpreter to remember the values in the earlier stack frame, even though they are not currently visible. The situation after creating the new frame and initializing the parameters looks like this:
5.3 The mechanics of function calls

The system then executes the statements in the function \texttt{fact}. In this instance, the body of the \texttt{for} loop is executed six times. On each cycle, the value of \texttt{result} is multiplied by the loop index \texttt{i}, which means that it will eventually hold the value 720 (\(1 \times 2 \times 3 \times 4 \times 5 \times 6\) or 6!). When the program reaches the \texttt{return} statement, the stack frame looks like this:

\begin{verbatim}
function combinations(n, k) {
  function fact(n) {
    let result = 1;
    for (let i = 1; i <= n; i++) {
      result *= i;
    }
    return result;
  } 
  return fact(n) / (fact(k) * fact(n - k));
}
\end{verbatim}

Returning from a function involves copying the value of the \texttt{return} expression (in this case the local variable \texttt{result}), into the point at which the call occurred. The frame for \texttt{fact} is then discarded, which leads to the following configuration:

\begin{verbatim}
function combinations(n, k) {
  return fact(n) / (fact(k) * fact(n - k));
}
\end{verbatim}

The next step in the process is to make a second call to \texttt{fact}, this time with the argument \texttt{k}. In the calling frame, \texttt{k} has the value 2. That value is then used to initialize the parameter \texttt{n} in the new stack frame, as follows:

\begin{verbatim}
function combinations(n, k) {
  return fact(n) / (fact(k) * fact(n - k));
}
\end{verbatim}
The computation of \( \text{fact}(2) \) is easier to perform in one’s head than the earlier call to \( \text{fact}(6) \). This time around, the value of \( \text{result} \) will be 2, which is then returned to the calling frame, like this:

\[
\text{function combinations}(n, k) \{
    \text{return } \frac{\text{fact}(n)}{(\text{fact}(k) \times \text{fact}(n - k))} ;
\}
\]

The code for \( \text{combinations} \) makes one more call to \( \text{fact} \), this time with the argument \( n - k \). Evaluating this call therefore creates a new stack frame with \( n \) equal to 4:

\[
\text{function fact}(n) \{
    \text{let } \text{result} = 1 ;
    \text{for } (\text{let } i = 1 ; i \leq n ; i++)
    \{
        \text{result } *= i ;
    \}
    \text{return } \text{result} ;
\}
\]

The value of \( \text{fact}(4) \) is \( 1 \times 2 \times 3 \times 4 \), or 24. When this call returns, the system is able to fill in the last of the missing values in the calculation, as follows:

\[
\text{function combinations}(n, k) \{
    \text{return } \frac{\text{fact}(n)}{(\text{fact}(k) \times \text{fact}(n - k))} ;
\}
\]

The computer then divides 720 by the product of 2 and 24 to get the answer 15. This value is then returned to the JavaScript interpreter running in the JavaScript console window. The interpreter prints that value on the console, like this:

```
JavaScript Console
> combinations(6, 2)  
15  
> 
```
5.4 Decomposition

One of the most important challenges you will face as a programmer is finding ways to reduce the conceptual complexity of your programs. Large programs are typically very difficult to understand as a whole. The only way to keep such programs within the limits of human comprehension is to break them up into simpler, more manageable pieces. This process is called decomposition.

You have already had a chance to apply the technique of decomposition in writing Karel programs, but the same strategy applies in JavaScript as well. In both languages, decomposition is the process of breaking large tasks down into simpler subtasks that together complete the task as a whole. Those subtasks may require further decomposition, which creates a hierarchy of subtasks of the sort illustrated in Figure 5-7. In that diagram—which presents only the general structure of a typical solution and offers no details about the problem itself—the complete task is decomposed into three primary subtasks. The second of those subtasks is then divided further into two subtasks at an even lower level of detail. Depending on the complexity of the actual problem, the subdivision may require more subtasks or more levels of decomposition.

Learning how to find the most useful decomposition requires considerable practice. If you define the individual subtasks appropriately, each one will have conceptual integrity as a unit and make the program as a whole much simpler to understand. If you choose the subtasks inappropriately, your decomposition can end up getting in the way. Although this chapter offers some useful guidelines,
there are no hard-and-fast rules for selecting a particular decomposition; you will learn how to apply this process through experience.

**Stepwise refinement**

When you are trying to find an effective decomposition strategy, the best approach is to start at the highest levels of abstraction and work your way downward to the details. You begin by thinking about the program as a whole. Assuming that the program is large enough to require decomposition, your next step is to divide the entire problem into its major components. Once you figure out what the major subtasks are, you can then repeat the process to decompose any of the subtasks that are themselves too large to solve in a few lines of code. At the end of this process, you will be left with a set of individual tasks, each of which is simple enough to be implemented as a single function. As you know from Chapter 1, this process is called *top-down design*, or *stepwise refinement*.

The best way to understand the process of stepwise refinement is to work through a simple example. Suppose that you want to write a graphical program that creates the following picture of a three-car train consisting of a black engine, a green boxcar, and a red caboose:

How would you go about designing such a program?

There are two important things to notice about this picture. The first is that it is composed entirely of shapes that you already know how to draw. Each of the cars consists of a filled `GRect`, as does the smokestack on the engine, the cab for the engineer, and the cupola sitting on top of the caboose. The doors on the boxcar are unfilled `GRects`. The wheels are `GOvals` filled in gray. The cowcatcher at the front of the engine and the connectors between the train cars are `GLines`. Since you know how to create the components of the picture, you could start off and write a `DrawTrain` function that added each of the graphical objects to the canvas with the appropriate location, size, and color. Before doing so, however, you might want to stop and think about possible strategies for simplifying the problem.

The second important thing to notice about the train picture is that, even though the diagram is more complicated than any of the graphical examples so far in this
book, it subdivides easily into smaller pieces. The task of drawing this particular train decomposes naturally into three subtasks: drawing an engine, drawing a boxcar, and drawing a caboose. Thus, if you want to write your DrawTrain function in a way that exploits the power of decomposition, you should subdivide it into three functions—drawEngine, drawBoxcar, and drawCaboose—each of which is responsible for drawing that type of car. This division of responsibility is illustrated in the following diagram:

Decomposing the program into functions has three advantages. First, this kind of subdivision makes the program easier to write by allowing you to focus on one part of the problem at a time. While writing drawEngine, for example, you can ignore the details of drawBoxcar and drawCaboose. Second, and perhaps more importantly, such decomposition makes the program easier to read, thereby reducing the conceptual complexity for programmers who may need to modify the code in the future. If nothing else, the function names make it clear what is going on in each part of the code. If you had instead combined all the individual steps necessary to create the picture into a single function, readers would have a much harder time figuring out which lines of code contributed to particular parts of the picture. Third, decomposition often supplies you with tools that you can reuse. In this simple train diagram, there is only one boxcar. In the real world, a freight train with only one boxcar would almost certainly run at a loss. To be economical, a single engine should pull many such cars. Having a drawBoxcar function allows you to draw as many boxcars as you need.

**Choosing the right parameters**

At this point, you know that it makes sense to decompose the problem of drawing a train into functions that draw each type of car. The next step in the process is to determine how those functions get the information they need to draw the picture correctly. In order to draw a picture of the sort shown in the sample run, each of the functions needs various pieces of information. For example, the drawBoxcar function needs to know the dimensions of the boxcar, what color it should be, and where to place the boxcar so that it correctly links up with the other cars.

There are three strategies you can use to provide such information to a function. The first is to pass information from the caller in the form of arguments. The second is to include that information explicitly as part of the program, typically by defining a constant that has the appropriate value. The third, which you will see in Chapter 6, is to define your functions in a hierarchical way so that lower-level
functions have access to the local variables declared in the functions defined at a higher level.

Adding more arguments offers greater flexibility but often makes it harder to understand how the function works. Whenever you call a function, you have to know what each of its arguments means. If drawing a boxcar requires you to specify the color of the boxcar, its location on the canvas, the size of the car itself, the size of the doors, and the radius of the wheels, you will probably have trouble keeping all those details straight. On the other hand, if you instead choose to define each of these parameters as a constant, you can draw a boxcar only in one color, in one size, and in one place.

To avoid the problems of each extreme, you need to strike a balance between these two strategies. As a general rule, it makes sense to apply the following guidelines in making that decision:

- Use parameters whenever callers are likely to supply different values for that parameter.
- Use constants when callers are likely to be satisfied with a value you’ve chosen.

In the case of the train diagram, the caller is probably not interested in controlling the radius of the wheels. That value should therefore be specified as a constant. At the other end of the spectrum, callers will certainly need to indicate where on the canvas a particular car should be placed. That information must therefore be conveyed through arguments. Other properties such as size and color fall between these extremes. For the purposes of this example, it is probably sufficient to make the following assumptions:

- All the cars are the same size.
- Engines are always black.
- Cabooses are always red.
- Boxcars come in many colors.

These assumptions initially suggest that the only arguments that the drawEngine and drawCaboose functions need are those that specify the location of the car and that the drawBoxcar function needs that same information along with the color of the boxcar. There is, however, an additional argument that each of these functions must have. Any function that works with the graphics window needs to have access to the GWindow object, which therefore needs to be passed as a parameter. Taking these considerations together, the header lines for these functions look like this:
function drawEngine(gw, x, y)
function drawBoxcar(gw, x, y, color)
function drawCaboose(gw, x, y)

The rest of the information necessary to create the picture can be specified using constants. For the train diagram shown in the sample run, these constants have the values shown in Figure 5-8.

As the header lines indicate, the `drawEngine`, `drawBoxcar`, and `drawCaboose` functions take an `x` and a `y` parameter to indicate the location of the car on the canvas. Callers that use these functions need to know what those coordinates mean in terms of where the car is drawn relative to the point `(x, y)`. There are many possible interpretations, and part of your job as programmer is to choose an appropriate design. One possibility is to adopt the convention used by the SJS graphics package and to define the point `(x, y)` to be the upper left corner of a car. In the absence of a compelling reason to the contrary, choosing a model that matches that of existing libraries is a good idea. In this case, however, there may indeed be a compelling reason. The location of the upper left corner of a train car depends on how tall the car is. In this example, the engine and the caboose have graphical figures on their top side that make them taller than the boxcar. Thus, to calculate the `y` coordinate of the top of a car, you would need to know the type of car. On the other hand, it is easy to calculate the `y` coordinate of the bottom of the car because all the cars are sitting on a track. Because all cars have a common baseline, it makes sense to let the `x` parameter indicate the left edge of the connector that extends from the car and to let the `y` parameter indicate the level of the track.

**Figure 5-8 Constants that control the appearance of the DrawTrain program**

```plaintext
const CAR_WIDTH = 113;            /* Width of the frame of a train car */
const CAR_HEIGHT = 54;             /* Height of the frame of a train car */
const CAR_BASELINE = 15;           /* Distance of car base to the track */
const CONNECTOR = 6;               /* Width of the connector on each car */
const WHEEL_RADIUS = 12;           /* Radius of the wheels on each car */
const WHEEL_INSET = 24;            /* Distance from frame to wheel center */
const CAB_WIDTH = 53;              /* Width of the cab on the engine */
const CAB_HEIGHT = 12;             /* Height of the cab on the engine */
const SMOKESTACK_WIDTH = 12;       /* Width of the smokestack */
const SMOKESTACK_HEIGHT = 12;      /* Height of the smokestack */
const SMOKESTACK_INSET = 12;       /* Distance from smokestack to front */
const DOOR_WIDTH = 27;             /* Width of the door on the boxcar */
const DOOR_HEIGHT = 48;            /* Height of the door on the boxcar */
const CUPOLA_WIDTH = 53;           /* Width of the cupola on the caboose */
const CUPOLA_HEIGHT = 12;          /* Height of the cupola on the caboose */
```
Designing from the top down

Now that you have decided on the headers for the `drawEngine`, `drawBoxcar`, and `drawCaboose` functions, it might seem as if the logical next step would be to go ahead and implement them. Although doing so is an option, it is usually better to complete the code at each level of decomposition before moving on to the next. By doing so, you can convince yourself that you have chosen the right decomposition. If you have left out a subtask or failed to include enough flexibility in the arguments to the functions, you will discover the problem when you try to code the highest level of the program. The principle of top-down design suggests that you should start with the highest-level function and then work your way down to the details.

The decomposition of the problem into subtasks makes the `DrawTrain` function easy to code. You know that the function will include calls to `drawEngine`, `drawBoxcar`, and `drawCaboose`. The only remaining issue is figuring out the coordinates to use in each call. If you look at picture in the sample run, you will see that the train is resting on the bottom of the canvas. The y coordinate of the track is therefore equal to the y coordinate of the bottom of the canvas, which you can find by calling `getHeights` on the `GWindow` object. Finding the x coordinate of each car requires a little more calculation. In this example, the entire train is centered in the window. Thus, calculating the x coordinate for the engine requires figuring out how long the entire train is and then subtracting half that length from the coordinates of the center of the canvas. Each subsequent car begins at a point whose x coordinate is shifted rightward by the width of a car and the width of its connector. Expressing these calculations in JavaScript gives rise to the following `DrawTrain` function:

```javascript
function DrawTrain() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let trainWidth = 3 * (CAR_WIDTH + 2 * CONNECTOR);
    let x = (gw.getWidth() - trainWidth) / 2;
    let y = gw.getHeight();
    let dx = CAR_WIDTH + 2 * CONNECTOR;
    drawEngine(gw, x, y);
    drawBoxcar(gw, x + dx, y, "Green");
    drawCaboose(gw, x + 2 * dx, y);
}
```

Looking for common features

Now that the `DrawTrain` function is out of the way, you can turn your attention to the functions that draw the individual cars. Although it is tempting to start typing in code, there is an enormous advantage in taking the time to think about the problem. When you consider possible decompositions, you should look for is subtasks that
 recur in multiple parts of the problem. To see how that strategy might apply in the current problem, it’s worth taking another look at the three different types of cars:

If you look carefully at the diagrams for these three cars, you will see that they share a number of common features. The wheels are the same, as are the connectors that link the cars together. In fact, the body of the car itself is the same except for the color. Each type of car shares a common framework that looks like this:

If you color the interior of the car with the appropriate color, you can use it as the foundation for any of the three car types. For the engine, you need to add a smokestack, a cab, and a cowcatcher. For the boxcar, you need to add doors. For the caboose, you need to add a cupola. In each case, much of the work can be delegated to a function that draws the common framework. That function—which itself decomposes into a function that draws each of the wheels—looks like this:

```javascript
function drawCarFrame(gw, x, y, color) {
    let xCar = x + CONNECTOR;
    let yCar = y - CAR_BASELINE;
    gw.add(GLine(x, yCar,
        x + CAR_WIDTH + 2 * CONNECTOR, yCar));
    drawWheel(gw, xCar + WHEEL_INSET, y - WHEEL_RADIUS);
    drawWheel(gw, xCar + CAR_WIDTH - WHEEL_INSET,
        y - WHEEL_RADIUS);
    let r = GRect(xCar, yCar - CAR_HEIGHT,
        CAR_WIDTH, CAR_HEIGHT);
    r.setFilled(true);
    r.setFillColor(color);
    gw.add(r);
}
function drawWheel(gw, x, y) {
    let r = WHEEL_RADIUS;
    let wheel = GOval(x - r, y - r, 2 * r, 2 * r);
    wheel.setFilled(true);
    wheel.setFillColor("Gray");
    gw.add(wheel);
}
Completing the decomposition

Once you have a tool for creating the framework of an individual car, you can complete the drawTrain program by filling in the definitions of the drawEngine, drawBoxcar, and drawCaboose functions. Some of the functions are easy enough to code without further decomposition. Here, for example, is an implementation of drawBoxcar that draws the background frame and then adds the two door panels:

```javascript
function drawBoxcar(gw, x, y, color) {
    drawCarFrame(gw, x, y, color);
    let xc = x + CONNECTOR + CAR_WIDTH / 2;
    let doorTop = y - CAR_BASELINE - DOOR_HEIGHT;
    gw.add(GRect(xc - DOOR_WIDTH, doorTop, DOOR_WIDTH, DOOR_HEIGHT));
    gw.add(GRect(xc, doorTop, DOOR_WIDTH, DOOR_HEIGHT));
}
```

You might, however, choose to decompose some of these functions further. For example, you could decide to break up the diagram of the train engine into its component parts and then code drawEngine like this:

```javascript
function drawEngine(gw, x, y) {
    drawCarFrame(gw, x, y, "Black");
    drawSmokestack(gw, x, y);
    drawEngineCab(gw, x, y);
    drawCowcatcher(gw, x, y);
}
```

Functions that tackle smaller parts of a larger problem, such as drawSmokestack, drawCab, and drawCowcatcher in this example, are often called helper functions.

If you adopt this decomposition strategy, you will have to write implementations for these helper functions along with the as-yet-unimplemented drawCaboose. You'll have a chance to finish the decomposition in exercise 9.

Summary

In this chapter, you learned about functions, which enable you to refer to an entire set of operations with a single name. More importantly, by allowing the programmer to ignore the internal details and concentrate only on the effect of a function as a whole, functions provide a critical tool for reducing the conceptual complexity of programs.

The important points introduced in this chapter include:
• A function consists of a set of program statements that have been collected together 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.

• A function that returns a value must have a return statement that specifies the result. Functions may return values of any type.

• Functions that return Boolean values are called predicate functions. Because you can use the result of such functions to specify a condition in an if or a while statement, predicate functions play an important role in programming.

• Variables declared within a function are local to that function and cannot be used outside it. Internally, all the variables declared within a function are stored together in a stack frame.

• Parameters are local variables that act as placeholders for the argument values. JavaScript initializes the parameter variables by copying the argument values in the order in which they appear. If there are more arguments than parameters, the extra ones are ignored; if there are more parameters than arguments, the extra parameters are initialized to the constant undefined.

• When a function returns, it continues from precisely the point at which the call was made. Computer scientists refer to this point as the return address.

• You can create your own libraries by collecting the necessary code in a file whose name ends with the standard .js file type. You can then use the entries exported by this library by including the appropriate <script> tag in the HTML file.

• In understanding the concept of a library, it is useful to differentiate the roles of the client, who uses the library, and the implementer, who writes the necessary code.

• Figure 5-2 shows the code for the RandomLib.js library, which provides a useful set of tools for working with programs that simulate random behavior.

• Functions are a powerful tool for reducing the complexity of programs because they allow you to decompose large tasks into smaller, more manageable subtasks.

• In most cases, it makes sense to decompose a program by starting at the level of the problem as a whole and then working your way downward to the details. This strategy is called top-down design or stepwise refinement.

Review questions

1. Define the following terms as they apply to functions: call, argument, return.
2. How do you specify the result of a function in JavaScript?
3. Can there be more than one return statement in the body of a function?
4. What is a predicate function?
5. Describe the differences between the roles of client and implementer.
6. How many functions are exported by the RandomLib.js library?
7. How do you gain access to the facilities provided by the RandomLib.js library from some other program file?
8. What is a half-open interval and how is such an interval usually represented in mathematics?
9. How would you use the randomInteger function to generate a randomly chosen integer between 1 and 100?
10. By going through the code by hand, determine whether the randomInteger function works with negative arguments. What are the possible results of calling the function randomInteger(-5, 5)?
11. If you run the RandomCircle.js program shown in Figure 5-4, you would expect to see 10 circles on the graphics window because N_CIRCLES has the value 10. In fact, you sometimes see fewer circles. Why might this be?
12. Describe the rules by which JavaScript assigns argument values to parameters.
13. Variables declared within a function are called local variables. What is the significance of the word local in this context?
14. What is a stack frame?
15. What do computer scientists mean by the term return address?
16. Explain in your own words the process of stepwise refinement.

Exercises

1. The Fibonacci sequence, in which each new term is the sum of the preceding two, was introduced in Chapter 4, exercise 10. Rewrite the program requested in that exercise, changing the implementation so that your program calls a function fib(n) to calculate the nth Fibonacci number. In terms of the number of mathematical calculations required, is your new implementation more or less efficient that the one you used in Chapter 4?
2. In contrast to most languages, JavaScript has few built-in facilities to support the creation of formatted tables, such as those in which numbers line up nicely in columns. To create this kind of formatted table, it is useful to have to create a library AlignLib.js that contains the functions alignLeft, alignRight (which appears in Chapter 4) and alignCenter, each of which takes a value and a width and returns the value aligned appropriately within a field of that size. In each case, you need to convert the value to a string, and then add spaces on the back, front, or alternately on both ends until the string has the desired length. You will have to make some decision as to how alignCenter operates if the number of extra spaces needed is odd. The comments associated with the function should document your decision.

3. Write a function randomAverage(n) that generates n random real numbers between 0 and 1 and then returns the average of those n values. Statistically, calling randomAverage(n) will produce results that become closer to 0.5 as the value of n increases. Write a main program that displays the result of calling randomAverage on 1, 10, 100, 1000, 10000, 100000, and 100000.

4. I shall never believe that God plays dice with the world.
   — Albert Einstein, 1947

Despite Einstein’s metaphysical objections, the current models of physics, and particularly of quantum theory, are based on the idea that nature does indeed involve random processes. A radioactive atom, for example, does not decay for any specific reason that we mortals understand. Instead, that atom has a probability of decaying randomly within a particular period of time.

Because physicists consider radioactive decay a random process, it is not surprising that random numbers can be used to simulate it. Suppose you start with a collection of atoms, each of which has a certain probability of decaying in any unit of time. You can then approximate the decay process by taking each atom in turn and deciding randomly whether it decays.

Write a function simulateRadioactiveDecay that models the process of radioactive decay. The first parameter is the initial population of atoms; the second is the probability that any of those atoms will decay within a year. For example, calling

simulateRadioactiveDecay(10000, 0.5)

simulates what happens over time to a sample that contains 10,000 atoms of some radioactive material, where each atom has a 50 percent chance of decaying in a year. Your function should produce a trace on the console showing how many atoms remain at the end of each year until all of the atoms have decayed. For example, the output of your function might look like this:
As the numbers indicate, roughly half the atoms in the sample decay each year. In physics, the conventional way to express this observation is to say that the sample has a **half-life** of one year.

5. Random numbers offer an interesting strategy for approximating the value of π. Imagine that you have a dartboard hanging on your wall that consists of a circle painted on a square backdrop, as in the following diagram:

What happens if you throw a whole bunch of darts completely randomly, ignoring any darts that miss the board altogether? Some of the darts will fall inside the pink circle, but some will be outside the circle in the white corners of the square. If the throws are random, the ratio of the number of darts landing inside the circle to the total number of darts hitting the square should be approximately equal to the ratio between the two areas. The ratio of the areas is independent of the actual size of the dartboard, as illustrated by the formula

\[
\frac{\text{area inside the circle}}{\text{area inside the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}
\]

To simulate this process in a program, imagine that the dart board is drawn on the standard Cartesian coordinate plane with its center at the origin and a radius of 1 unit. The process of throwing a dart randomly at the square can be modeled by generating two random numbers, \(x\) and \(y\), each of which lies between \(-1\) and \(+1\). This \((x, y)\) point always lies somewhere inside the square. The point \((x, y)\) lies inside the circle if
This condition, however, can be simplified considerably by squaring each side of the inequality, which yields the following more efficient test:

\[
x^2 + y^2 < 1
\]

If you perform this simulation many times and compute what fraction of the darts fall inside the circle, the result will be an approximation of \(\pi/4\).

Write a program that simulates throwing 10,000 darts and then uses the simulation technique described in this exercise to generate and display an approximate value of \(\pi\). Don’t worry if your answer is correct only in the first few digits. The strategy used in this problem is not particularly accurate, even though it occasionally proves useful as an approximation technique. In mathematics, this technique is called Monte Carlo integration, after the capital city of Monaco, famous for its casinos.

6. Write a function `consecutiveHeads(numberNeeded)` that simulates tossing a coin repeatedly until the specified number of heads appear consecutively. At that point, your program should display a line on the console that indicates how many coin tosses were needed to complete the process. The following console log shows one possible execution of the program:

```
> consecutiveHeads(3);
Tails
Tails
Heads
Tails
Heads
Heads
It took 8 tosses to get 3 consecutive heads.
```

7. The values of the combinations function \(C(n, k)\) described in this chapter are often displayed in the form of a triangle using a triangular arrangement that begins
and then continues for as many rows as desired. This figure is called **Pascal’s Triangle** after its inventor, the seventeenth-century French mathematician Blaise Pascal. Pascal’s Triangle has the interesting property that every interior entry is the sum of the two entries above it.

Write a function `displayPascalTriangle(n)` that displays Pascal’s Triangle from row 0 up to row `n`, as shown in the following console log:

```
> displayPascalTriangle(9);
    1
   1 1
  1 2 1
 1 3 3 1
 1 4 6 4 1
 1 5 10 10 5 1
 1 6 15 20 15 6 1
 1 7 21 35 35 21 7 1
 1 8 28 56 70 56 28 8 1
```

The interesting challenge in this assignment is aligning the output, for which the various functions you wrote for exercise 2 will come in handy.

8. The function \( C(n, k) \) determines the number of ways you can choose \( k \) values from a set of \( n \) elements, ignoring the order of the elements. If the order of the value matters—so that, in the case of the coin example, choosing a quarter first and then a dime is seen as distinct from choosing a dime and then a quarter—you need to use a different function, which computes the number of **permutations**, which are all the ways of ordering \( k \) elements taken from a collection of size \( n \). This function is denoted as \( P(n, k) \), and has the following mathematical formulation:

\[
P(n, k) = \frac{n!}{(n-k)!}
\]

Although this definition is mathematically correct, it is not well suited to implementation in practice because the factorials involved quickly get much too large. For example, if you use this formula to calculate the number of ways
to select two cards from a standard 52-card deck, you would end up trying to evaluate the following fraction:

\[
\frac{80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,000,000,000,
30,414,093,201,713,378,043,612,608,166,064,768,844,377,641,568,960,512,000,000,000,000}{52 \times 51}
\]

even though the answer is the much more manageable 2652 (52 \times 51).

Write a function \texttt{permutations(n, k)} that computes the \(P(n, k)\) function without calling the \texttt{fact} function. Part of your job in this problem is to figure out how to compute this value efficiently. To do so, you will probably find it useful to play around with some relatively small values to get a sense of how the factorials in the numerator and denominator of the formula behave.

9. Complete the implementation of the \texttt{DrawTrain.js} program by supplying the missing functions (\texttt{drawSmokestack}, \texttt{drawCab}, \texttt{drawCowcatcher}, and \texttt{drawCaboose}).

10. When you write a check, it is conventional to record the amount of the check both in figures and in words. Thus, if you were writing a check for $1,729 dollars, you would write the amount out in words as \textit{one thousand seven hundred and twenty-nine}. Implement a function

\[
\text{function convertNumberToWords(n)}
\]

that takes a number—which you may assume is an integer between 0 and 999,999—and returns a string that represents that number in words. Your function should be able to generate the following console session:

```javascript
> convertNumberToWords(17)
seventeen
> convertNumberToWords(2001)
two thousand one
> convertNumberToWords(136211)
one hundred thirty-six thousand two hundred eleven
```

This exercise requires no string operations other than concatenation and is intended primarily to give you practice with stepwise refinement. It is, however, important to note that there is no magical way to convert a digit like 1 into the English word \textit{one}. To implement that correspondence, you need to include explicit code—probably in the form of a \texttt{switch} statement—that converts each digit to its English equivalent. Think carefully about how to decompose the problem so that you can reuse the same methods to translate different parts of the number.
11. In the 1930s, the American psychologist J. Ridley Stroop conducted a series of experiments showing that people find it easier to read color names when the name matches the color in which it appears than when the name and its color interfere. Write a graphics program that randomly positions seven labels on the graphics window whose names correspond to the seven colors of the rainbow: "Red", "Orange", "Yellow", "Green", "Blue", "Indigo", and "Violet". The colors in which these labels are displayed, however, should be shifted three colors ahead in the spectrum, wrapping around, if necessary, at the end. The "Red" label therefore appears in green, and the "Indigo" label appears in orange, as shown in the following sample run:
CHAPTER 6

Writing Interactive Programs

Quit worrying about failure. Failure’s easy. Worry about if you’re successful, because then you have to deal with it.

— Adele Goldberg, interview with John Mashey, 2010

Adele Goldberg (1945–)

Adele Goldberg received her Ph.D. in Information Science from the University of Chicago and took a research position at the Xerox Palo Alto Research Center (PARC), which introduced the graphical user interface—an idea that has since become central to modern computing. Together with others in the Learning Research Group at PARC, Goldberg designed and implemented the programming language Smalltalk, which took the ideas of object-oriented programming developed in Scandinavia and integrated them into a programming environment designed to support constructivist learning. Drawing on the state-of-the-art technology invented at PARC, Smalltalk was among the first programming environments designed for use with graphical displays. Along with her colleagues Alan Kay and Dan Ingalls, Goldberg received the Software Systems Award from the Association for Computing Machinery, the leading professional society for computer science, in 1987.
So far, the only direct interactions you have had with JavaScript programs have taken place in the context of the JavaScript console. When you type a function call into the console window, the JavaScript interpreter then calls that function and displays the result. That style of interaction is synchronous, because user actions are synchronized with the program operation. A graphical user interface (often shortened to the acronym GUI, which is pronounced like gooey), by contrast, is asynchronous, in that it allows the user to intercede at any point, typically by using the mouse or the keyboard to trigger an action. Actions that occur asynchronously with respect to the program operation, such as clicking the mouse or typing on the keyboard, are generically referred to as events. Interactive programs that operate by responding to these events are said to be event-driven. The primary goal of this chapter is to teach you how to write simple event-driven programs.

Historically, the development of the graphical user interface has been closely associated with the object-oriented paradigm, which is itself commonly abbreviated as OOP. There are at least two reasons that the GUI and OOP have worked well together (beyond the fact that they have both become popular three-letter buzzwords in the computing industry). First, graphical displays are characterized by having many independent objects that form a hierarchical relationship that fits easily into the inheritance hierarchy that object-oriented languages provide. Second, it is easy to think of events as messages, which are a central foundation of the object-oriented model. Clicking the mouse, for example, sends a message to the application, which then responds in an appropriate way.

6.1 First-class functions

Before looking at the details of how event-driven programs are implemented in JavaScript, it is useful to spend a little more time looking at the question of how JavaScript implements the idea of a function. In the programs you have seen so far in this book, the ideas of functions and data have remained separate. Functions provide the means for representing an algorithm; those functions then operate on data values, which act as the raw material on which computation is performed. Functions have been part of the algorithmic structure, not part of the data structure. Being able to use functions as data values, however, often makes it much easier to design effective interfaces, because this facility allows clients to specify operations as well as data.

In JavaScript, functions are values that are simultaneously part of both the algorithmic structure and the data structure of a program. Given a functional value, you can assign it to a variable, pass it as a parameter, or return it as a result. When a programming language allows functions to behave just like any other data value, computer scientists say that the language supports first-class functions.
Declaring functions as data values

In JavaScript, the style of function definition that you have seen is shorthand for declaring a variable with the function as its value. For example, instead of writing

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

JavaScript allows you to achieve the same result through the following declaration:

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

This declaration introduces a new variable named `f` and initializes that variable to a function that takes an argument `x` and returns the value \( x^2 - 5 \). The syntax matches that of any other JavaScript declaration and therefore takes a semicolon at the end.

There is no particular advantage in using this style of declaration, but the fact that you can assign a functional value to a variable will certainly play an important role in this chapter.

The domain of the function data type is the vast spectrum of functions that you might think to define in JavaScript. The operation that is particular to the function data type is **application**, which is the process of calling that function with a list of arguments. No matter which way you define the function `f`, you call it in the same way, so that `f(3)` produces the value 4.

In JavaScript, a function defined inside another function is called an **inner function**. The primary advantage of inner functions is that they have access to the local variables of the function in which the inner function is defined. The value of an inner function is therefore more than just the code that implements it. Inner functions also keep track of the variables that are defined in the frame of the enclosing function. This combination of code and variables is called a **closure**.

Passing functions as parameters

One example of an application in which passing functions as parameters makes intuitive sense is the following function:

```javascript
function printFunctionTable(f, min, max) {
    for (let i = min; i <= max; i++) {
        console.log("f(" + i + ") = " + f(i));
    }
}
```

This first parameter in calls to `printFunctionTable` is a function that takes a number and returns a result, which is usually a number but need not be for certain
applications. The effect of this function is to count from $\text{min}$ to $\text{max}$, generating a line of output showing the value of the function at each of those values. For example, if $f$ is defined as shown in the last section to be $f(x) = x^2 - 5$, calling

```javascript
printFunctionTable(f, -4, 4);
```

produces the following output:

```
JavaScript Console
f(-4) = 11
f(-3) = 4
f(-2) = -1
f(-1) = -4
f(0) = -5
f(1) = -4
f(2) = -1
f(3) = 4
f(4) = 11
```

The function argument, however, can be any function, even one that comes from a library. For example, calling

```javascript
printFunctionTable(Math.sqrt, 0, 9);
```

generates the following console log:

```
JavaScript Console
f(0) = 0
f(1) = 1
f(2) = 1.4142135623730951
f(3) = 1.7320508075688772
f(4) = 2
f(5) = 2.23606797749979
f(6) = 2.449489742783178
f(7) = 2.64575131106645907
f(8) = 2.8284271224481903
f(9) = 3
```

It is also legal in JavaScript to specify a function definition directly within the function call, as in

```javascript
printFunctionTable(function(x) { return x * x; }, 1, 5);
```

which displays the following table of squares:

```
JavaScript Console
f(1) = 1
f(2) = 4
f(3) = 9
f(4) = 16
f(5) = 25
```
6.2 A simple interactive example

Before getting too deeply buried in the details, it helps to consider a simple example that illustrates JavaScript’s interaction model. The DrawDots.js program in Figure 6-1 draws a small dot whenever the user clicks the mouse button. For example, if you click the mouse near the upper left corner of the window, the program will draw a dot in that position, as shown in the following diagram:

```javascript
/*
 * File: DrawDots.js
 * --------------------------------
 * This program draws a dot every time the user clicks the mouse.
 */

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const DOT_SIZE = 6;

/* Main program */
function DrawDots() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);

    function clickAction(e) {
        let dot = GOval(e.getX() - DOT_SIZE / 2, e.getY() - DOT_SIZE / 2,
                        DOT_SIZE, DOT_SIZE);
        dot.setFill(true);
        gw.add(dot);
    }

    gw.addEventListener("click", clickAction);
}
```
If you then go on to click the mouse in other positions, dots will appear there as well. You could, for example, draw a picture of the constellation Ursa Major, which is more commonly known as the Big Dipper. All you have to do is click the mouse once in the position of each star, as follows:

![Drawing of constellation](image)

The code in Figure 6-1 is extremely short, with just a single top-level function along with a few constant definitions. Even so, the program is different enough from the ones that you’ve seen so far that it makes sense to go through it in detail.

The first statement in the program simply creates the graphics window, precisely as you always have. The code then defines the function `clickAction`, which is described in more detail in the following paragraph. The last statement establishes the link between the graphics window and the behavior specified by the `clickAction` function. By executing the line

```
gw.addEventListener("click", clickAction);
```

the program is telling the graphics window that it wants to respond to mouse clicks. Moreover, the response to that mouse click is specified by `clickAction`, which should be called whenever a click occurs.

The function `clickAction` is never called explicitly in the code shown in Figure 6-1. The call, when it happens, comes from the code that implements the graphics library. Functions that are not called directly but are instead invoked in response to some event are referred to as callback functions. The name reflects the relationship between the client program and the libraries it uses. As a client, your program calls `addEventListener` to register interest in a particular event. As part of that process, you provide the library with a function that it can call when the event occurs. It is, in a way, analogous to providing a callback number. When the library implementation needs to call you back, you’ve given it the means to do so.
It now makes sense to look more carefully at the `clickAction` function:

```javascript
function clickAction(e) {
    let dot = GOval(e.getX() - DOT_SIZE / 2,
                    e.getY() - DOT_SIZE / 2,
                    DOT_SIZE, DOT_SIZE);
    dot.setFilled(true);
    gw.add(dot);
}
```

The function takes a parameter `e`, which provides the function with data about the details of the event. In this case, `e` is a *mouse event*, which keeps track of such information as the location of the mouse. Mouse events in JavaScript contain more data, but the location of the mouse can be determined by invoking the methods `e.getX()` and `e.getY()`. Each of these methods returns a coordinate in pixels measured relative to the origin in the upper left corner of the window.

The body of the `clickAction` function creates a `GOval` of the size specified by `DOT_SIZE`, sets it to be filled, and then adds it to the window so that its center appears as the current mouse position. The variable `gw`, which is a local variable inside `DrawDots`, is accessible to the `clickAction` code because that function is defined inside the `DrawDots` body.

### 6.3 Controlling properties of objects

Before moving on to look at more sophisticated examples of interactivity, it is important to have a better understanding of how to manipulate graphical objects that have already been placed on the screen. So far, the objects that you have added to the graphics window retain their initial location and dimensions. When you build interactive programs, you need to be able to change these properties.

The `GObject` class exports a much richer set of methods than you have had a chance to use so far. Figure 6-2 lists the complete set of methods supported by the `GObject` class and its `GRect`, `GOval`, and `GLine` subclasses. Each of the method descriptions consists of a single line that offers an overview of what the method does. For more details, you will need to consult the web documentation for the graphics library.

Instead of going through each of these methods in detail, it is more productive to work through several programming examples and then introduce whatever new methods are necessary to make each example work. That way, you see each of these methods in the context of an application that makes use of it.
## Expanded set of methods available in the GObject class

### Methods that control the location of the object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.setLocation(x, y)</code></td>
<td>Sets the location of this object to the point ((x, y)).</td>
</tr>
<tr>
<td><code>obj.move(dx, dy)</code></td>
<td>Moves the object using the displacements (dx) and (dy).</td>
</tr>
<tr>
<td><code>obj.movePolar(r, theta)</code></td>
<td>Moves the object (r) pixels in direction (theta).</td>
</tr>
</tbody>
</table>

### Methods that control the appearance of the object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.setColor(color)</code></td>
<td>Sets the color used to display this object.</td>
</tr>
<tr>
<td><code>obj.setLineWidth(width)</code></td>
<td>Sets the width of the lines used to draw the object.</td>
</tr>
<tr>
<td><code>obj.setVisible(flag)</code></td>
<td>Sets whether this object is visible.</td>
</tr>
<tr>
<td><code>obj.rotate(theta)</code></td>
<td>Rotates the object (theta) degrees around its origin.</td>
</tr>
<tr>
<td><code>obj.scale(sf)</code></td>
<td>Scales the object by (sf) both horizontally and vertically.</td>
</tr>
<tr>
<td><code>obj.scale(sx, sy)</code></td>
<td>Scales the object by (sx) horizontally and (sy) vertically.</td>
</tr>
</tbody>
</table>

### Methods that control the stacking order

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.sendBackward()</code></td>
<td>Moves this object one step backward in the stacking order.</td>
</tr>
<tr>
<td><code>obj.sendForward()</code></td>
<td>Moves this object one step forward in the stacking order.</td>
</tr>
<tr>
<td><code>obj.sendToBack()</code></td>
<td>Moves this object to the back of the stacking order.</td>
</tr>
<tr>
<td><code>obj.sendToFront()</code></td>
<td>Moves this object to the front of the stacking order.</td>
</tr>
</tbody>
</table>

### Methods that return properties of the object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.getX()</code></td>
<td>Returns the (x) coordinate of the object.</td>
</tr>
<tr>
<td><code>obj.getY()</code></td>
<td>Returns the (y) coordinate of the object.</td>
</tr>
<tr>
<td><code>obj.getWidth()</code></td>
<td>Returns the width of this object.</td>
</tr>
<tr>
<td><code>obj.getHeight()</code></td>
<td>Returns the height of this object.</td>
</tr>
<tr>
<td><code>obj.getColor()</code></td>
<td>Returns the color used to display this object.</td>
</tr>
<tr>
<td><code>obj.getLineWidth()</code></td>
<td>Returns the width of the lines used to draw the object.</td>
</tr>
<tr>
<td><code>obj.isVisible()</code></td>
<td>Checks to see whether this object is visible.</td>
</tr>
<tr>
<td><code>obj.contains(x, y)</code></td>
<td>Checks to see whether the point ((x, y)) is inside the object.</td>
</tr>
</tbody>
</table>

### Methods available only for the GRect and G Oval classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.setFill(flag)</code></td>
<td>Sets whether this object is filled.</td>
</tr>
<tr>
<td><code>obj.setFillColor(color)</code></td>
<td>Sets the color used to fill the interior of the object.</td>
</tr>
<tr>
<td><code>obj.setBoundaries(x, y, width, height)</code></td>
<td>Resets the boundary rectangle for the object.</td>
</tr>
</tbody>
</table>

### Methods available only for the GLine class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.setStartPoint(x, y)</code></td>
<td>Changes the start point of the line without changing the end.</td>
</tr>
<tr>
<td><code>obj.setEndPoint(x, y)</code></td>
<td>Changes the end point of the line without changing the start.</td>
</tr>
</tbody>
</table>
6.4 Responding to mouse events

The "click" event used in the DrawDots.js program is only one of several mouse events that JavaScript allows you to detect. The mouse events implemented by the GWindow class are shown in Figure 6-3. Each of these event names allows you to respond to a specific type of action with the mouse, most of which will seem familiar from using your computer. The "mousemove" event, for example, is generated when you move the mouse in the window without pressing the mouse button. The "drag" event occurs when you move the mouse while holding the button down. The name of the event comes from the fact that the interaction model of moving the mouse with the button down is often used to drag objects around on the window. You press the mouse button over an object to grab it and then drag it to the desired position.

The sections that follow offer several examples that illustrate conventional styles of using the mouse to create and reposition objects in the graphics window.

A simple line-drawing program

In all likelihood, you have already used some application that allows you to draw lines on the screen by dragging the mouse. To create a line, you press the mouse button at the point at which you’d like the line to start and then drag the mouse with the button down until you reach the point at which the line ends. As you drag the mouse, the application typically updates the line so that you can see what you have drawn so far. When you release the mouse button, the line stays in that position, and you can repeat the process to create as many new lines as you want.

Suppose, for example, that you press the mouse button somewhere on the screen and then drag the mouse rightward an inch, holding the button down. What you’d like to see is the following picture:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;click&quot;</td>
<td>The user clicks the mouse in the window.</td>
</tr>
<tr>
<td>&quot;dblclick&quot;</td>
<td>The user double-clicks the mouse in the window.</td>
</tr>
<tr>
<td>&quot;mousedown&quot;</td>
<td>The user presses the mouse button.</td>
</tr>
<tr>
<td>&quot;mouseup&quot;</td>
<td>The user releases the mouse button.</td>
</tr>
<tr>
<td>&quot;mousemove&quot;</td>
<td>The user moves the mouse with the button up.</td>
</tr>
<tr>
<td>&quot;drag&quot;</td>
<td>The user drags the mouse (i.e., moves the mouse with the button down).</td>
</tr>
</tbody>
</table>
If you then move the mouse downward without releasing the button, the displayed line will track the mouse, so that you might see the following picture:

As you drag the mouse, the application repeatedly updates the line, making it appear to stretch as the mouse moves. Because the effect is precisely what you would expect if you joined the starting point and the mouse cursor with a stretchy elastic line, this technique is called rubber-banding.

When you release the mouse, the line stays where it is. If you then press the mouse button again on that same point, you can go ahead and draw an additional line segment by dragging the mouse to the end point of the new line, as follows:

The code for this application, which appears in Figure 6-4, is remarkably short, despite the fact that the program performs what seems like a sophisticated task. As in the DrawDots program, the creation of the GWindow object at the beginning of the program follows the standard convention. The program then declares a variable called line that will keep track of the current line. When the program starts, there is no line, and the program records that fact by setting its initial value to the special
value `null`, which is used in JavaScript to indicate an object that doesn’t currently exist. The value of `line` is set in the code for the “mousedown” event and updated in the code for the “drag” event. The fact that it is set in one function and updated in another means that `line` has to be declared in the `DrawLines` function so that both of the event-handling functions have access to it.

The function `mousedownAction` creates a new `GLine` object, assigns it to the variable `line`, and then adds the line to the window. Initially, the line starts and ends at the current position of the mouse, which means that it appears as a dot. The function `dragAction` calls the `setEndPoint` method in the `GLine` class, which, as noted in Figure 6-2, changes the point at which the line ends without changing its starting point. Doing so produces the desired rubber-banding behavior.

### Dragging objects on the canvas

The `DragObjects.js` program in Figure 6-5 offers a slightly more sophisticated example of an event-driven program that uses the mouse to reposition objects on the
/*
 * File: DragObjects.js
 * _______________________
 * This program illustrates how to drag objects on the window.
 */

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 200;
const GOBJECT_WIDTH = 200;
const GOBJECT_HEIGHT = 100;

/* Main program */
function DragObjects() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let x0 = (gw.getWidth() - GOBJECT_WIDTH) / 2;
    let y0 = (gw.getHeight() - GOBJECT_HEIGHT) / 2;
    let rect = GRect(x0, y0, GOBJECT_WIDTH, GOBJECT_HEIGHT);
    rect.setFill(true);
    rect.setStroke("Blue");
    gw.add(rect);
    let oval = GOval(x0, y0, GOBJECT_WIDTH, GOBJECT_HEIGHT);
    oval.setFill(true);
    oval.setStroke("Red");
    gw.add(oval);
    let lastX = 0;
    let lastY = 0;

    function mousedownAction(e) {
        lastX = e.getX();
        lastY = e.getY();
        objectBeingDragged = gw.getElmAt(lastX, lastY);
    }

    function dragAction(e) {
        if (objectBeingDragged !== null) {
            objectBeingDragged.move(e.getX() - lastX, e.getY() - lastY);
            lastX = e.getX();
            lastY = e.getY();
        }
    }

    function clickAction(e) {
        if (objectBeingDragged !== null) {
            objectBeingDragged.sendToFront();
        }
    }

    gw.addEventListener("mousedown", mousedownAction);
    gw.addEventListener("drag", dragAction);
    gw.addEventListener("click", clickAction);
}
display. This program begins by creating a blue rectangle and a red oval on the window, just as in the `GRectPlusGOval.js` program from Chapter 3. The rest of the main program represents the code pattern for dragging objects.

The first callback function defined in Figure 6-5 is `mousedownAction`, which is called when the mouse button goes down. Here, the `mousedownAction` function has the following definition:

```java
function(e) {
  lastX = e.getX();
  lastY = e.getY();
  objectBeingDragged = gw.getElementAt(lastX, lastY);
}
```

The first two statements simply record the x and y coordinates of the mouse in the variables `lastX` and `lastY`. As you can see from the program, these variables are declared as local variables in the enclosing `DragObjects` function, because the program needs these values in the callback function that responds when the mouse is dragged.

The last statement in the `mousedownAction` function makes use of an important new method in the `GWindow` class. The `getElementAt` method takes an x and a y coordinate and then checks to see what object displayed on the window contains that location. Here, it is important to recognize that there are two possibilities. First, you could be pressing the mouse button on top of an object, which means that you want to start dragging it. Second, you could be pressing the mouse button somewhere else on the canvas at which there is no object to drag. If there is just one object at the specified location, `getElementAt` returns that object; if more than one object covers that space, `getElementAt` chooses the one that is in front of the others in the stacking order. If there are no objects at that location, `getElementAt` returns the special value `null`, which signifies an object that does not exist. In any of those cases, the `mousedownAction` function assigns that value to the variable `objectBeingDragged`, which is again declared as a level of the `DragObjects` function so that it can be shared with other callback functions.

The function assigned to `dragAction` consists of the following code:

```java
function(e) {
  if (objectBeingDragged !== null) {
    objectBeingDragged.move(e.getX() - lastX, e.getY() - lastY);
    lastX = e.getX();
    lastY = e.getY();
  }
}
```
The `if` statement checks to see whether there is an object to drag. If the value of `objectBeingDragged` is `null`, there is nothing to drag, so the rest of the function can just be skipped. If there is an object, you need to move it by some distance in each direction. That distance does not depend on where the mouse is in an absolute sense but rather on how far it has moved from where you last took stock of its position. Thus, the arguments to the `move` method are—for both the `x` and `y` components—the location where the mouse is now minus where the mouse was at the time of the last event. Those coordinates are stored in the variables `lastX` and `lastY`. Once you have moved the object, you then have to update the values of these variables to ensure that they are correct for the next call to `mouseDragged`.

The only other feature in the `DragObjects.js` program is that the application also registers its interest in "click" events, which trigger a call to the following function:

```javascript
function(e) {
    if (objectBeingDragged !== null) {
        objectBeingDragged.sendToFront();
    }
}
```

The point of adding this function is to allow the user to change the stacking order, which, as noted in Chapter 2, is the order in which objects are layered on the screen. In the `DragObjects.js` program, clicking on an object has the effect of moving it to the front of the stacking order. This function, however, depends on JavaScript’s rules for generating mouse events. A "click" event occurs when a "mousedown" event is followed within a relatively short amount of time by a "mouseup" event. By the time the "click" event is processed, JavaScript has already taken whatever actions are specified for both the "mousedown" and "mouseup" events. The `DragObjects.js` program does not specify any action for "mouseup" but does respond to the "mousedown" event by calling the `mousedownAction` function. Thus, by the time `clickAction` is called, the `mousedownAction` function will already have set the value of `objectBeingDragged`.

### 6.5 Timer-based animation

Interactive programs change their behavior not only in response to user events, but also over time. In a computer game, for example, objects on the screen typically move in real time. Updating the contents of the graphics window so that they change over time is called animation.
The `setTimeout` and `setInterval` functions

The conventional way to implement animation in JavaScript is to use a timer, which is a mechanism that generates a function call after a specified delay. JavaScript timers come in two forms. The library function

```javascript
setTimeout(function, delay)
```

creates a one-shot timer that calls `function` after `delay` milliseconds. The function

```javascript
setInterval(function, delay)
```

creates an interval timer that calls `function` repeatedly every `delay` milliseconds. In each case, the function returns a numeric value that identifies the timer. If you store that value in a variable, you can then call `clearTimeout` or `clearInterval` (depending on the type of timer) to stop the timer process. Thus, executing

```javascript
let timer = setInterval(step, 20);
```

creates an interval timer and stores its identifying number in the variable `timer`. The interval timer then begins generating calls to the function `step` once every 20 milliseconds, or every fiftieth of a second. The name `step` is chosen here to suggest that each call represents a single step in the animation, which is called a time step. The `step` function receives no arguments, so any information it needs must be communicated through the local variables of the function in which `step` is defined.

Timers that initiate events every 20 milliseconds allow you to change the state of the graphics window quickly enough so that the changes seem smooth to the human eye. You can therefore move an object on the screen by creating an interval timer that executes its callback function every 20 milliseconds and then having the callback function make an incremental change to the position of that object. When the object reaches the desired final location, your program can then stop the timer by calling

```javascript
clearInterval(timer);
```

A simple example of animation

A simple example of timer-based animation appears in Figure 6-6, which moves a square diagonally across the screen from its initial position in the upper left corner to its final position in the lower right. The program runs for `N_STEPS` time steps and computes values for the variables `dx` and `dy` so that the square moves to its final position in precisely that amount of time.
/*
 * File: AnimatedSquare.js
 * ----------------------
 * This program animates a square so that it moves from the upper left corner of the window to the lower right corner.
 */

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const N_STEPS = 100;
const TIME_STEP = 20;
const SQUARE_SIZE = 50;

/* Main program */

function AnimatedSquare() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let dx = (gw.getWidth() - SQUARE_SIZE) / N_STEPS;
    let dy = (gw.getHeight() - SQUARE_SIZE) / N_STEPS;
    let square = GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
    square.setFilled(true);
    gw.add(square);
    let stepCount = 0;
    function step() {
        square.move(dx, dy);
        stepCount++;
        if (stepCount == N_STEPS) clearInterval(timer);
    }
    let timer = setInterval(step, TIME_STEP);
}
The code for the callback function looks like this:

```javascript
function step() {
    square.move(dx, dy);
    stepCount++;
    if (stepCount === N_STEPS) clearInterval(timer);
}
```

The first line adjusts the position of the square by the values `dx` and `dy`. The second line increments the value of `stepCount`. The third line checks to see whether `stepCount` has reached the limit and, if so, stops the timer. The `step` function has access to the variables `square`, `dx`, `dy`, `stepCount`, and `timer` because these are local variables in `AnimatedSquare`.

If you look carefully at the code in Figure 6-6, you may notice a feature that at first seems a little odd. The `step` function is defined before the code creates the `timer`. That ordering is essential because the `setInterval` function takes the callback function as one of its arguments, which therefore must be initialized at the time of the call. What might seem confusing is that the `step` function refers to the variable `timer` even though that variable is declared after the definition of `step`. How, you may ask, does `step` have access to the local variable `timer` if it has not yet been declared?

The answer to this question lies in the fact that a closure in JavaScript includes all the variables declared within a block at the time the closure is used. The best way to think about this model is that the closure is tied to the pair of curly braces that surround the function definition and includes all the names defined anywhere within those braces, even if those names are declared after the function itself. In programming, the collection of names defined inside a block is called a `scope`. The definition of the `step` function stores the current scope as part of the closure. When the declaration of the variable `timer` appears immediately after the definition of the `step` function, the local variable `timer` is added to the current scope, which is stored as part of the closure. Since the closure is a shared object, the variable `timer` is accessible to the `step` function, just as it needs to be.

**Tracking the state of an animation**

As animations become more complex, keeping track of what objects are moving on the screen and when the next time step should occur becomes a bit tricky. Suppose, for example, that you want to add animation to the `RandomCircles.js` program that appears in Figure 5-4 on page 158. Instead of having the circles all appear on the screen at once, what you want instead is for the circles to appear slowly one at a time. Each circle first appears as a single point and then grows, step by step, until it reaches its desired size. As soon as that happens, the program should create the
next circle and let it grow to its full size, continuing in this fashion until all ten
circles are displayed on the screen.

It is, of course, tempting to start this program by building on the earlier example.
That strategy would suggest adopting the following pseudocode structure inside the
main program:

```javascript
for (let i = 0; i < N_CIRCLES; i++) {
  Create a circle.
  Animate that circle so that it grows to full size.
  Wait for that animation to complete.
}
```

Unfortunately, that strategy doesn’t work in JavaScript. Unlike most languages,
JavaScript does not allow programs to wait for some asynchronous task to finish
and then continue with what they were doing. That restriction makes it impossible
to capture the intention of the pseudocode line

*Wait for that animation to complete.*

Interactivity in JavaScript is required to be entirely *event-driven*, in the sense
that all actions take place in response to events that occur asynchronously with
respect to the running of the program. In fact, programs in JavaScript typically run
to completion before any events occur. After that, events completely determine
how the application proceeds. JavaScript’s event model requires a different
approach, in which the `step` function that implements the animation has to keep
track of the size of the current circle and determine when it has reached full size.
When it has, the `step` function—and not the main program, which has stopped
running by this point—must create the next circle. The `step` function therefore has
the following pseudocode form:

```javascript
function step() {
  if (the current circle is still growing) {
    Increase the size of the current circle.
  } else if (there are more circles to create) {
    Create another circle.
  } else {
    clearInterval(timer);
  }
}
```

The code for the `GrowingCircles.js` program that uses this structure appears in
Figure 6-7.
/*
 * File: GrowingCircles.js
 * ---------------------
 * This program draws random circles that grow to their final size.
 */

/* Constants */

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const N_CIRCLES = 10;
const MIN_RADIUS = 15;
const MAX_RADIUS = 50;
const TIME_STEP = 20;
const DELTA_SIZE = 1;

/* Main program */

function GrowingCircles() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let circlesCreated = 0;
    let desiredSize = 0;
    let currentSize = 0;
    let circle = null;

    function createNewCircle() {
        let r = randomReal(MIN_RADIUS, MAX_RADIUS);
        let x = randomReal(r, GWINDOW_WIDTH - r);
        let y = randomReal(r, GWINDOW_HEIGHT - r);
        circle = GOval(x, y, 0, 0);
        circle.setFilled(true);
        circle.setColor(randomColor());
        desiredSize = 2 * r;
        currentSize = 0;
        return circle;
    }

    function step() {
        if (currentSize < desiredSize) {
            currentSize += DELTA_SIZE;
            let x = circle.getX() - DELTA_SIZE / 2;
            let y = circle.getY() - DELTA_SIZE / 2;
            circle.setBounds(x, y, currentSize, currentSize);
        } else if (circlesCreated < N_CIRCLES) {
            gw.add(createNewCircle());
            circlesCreated++;
        } else {
            clearInterval(timer);
            return;
        }
    }

    let timer = setInterval(step, TIME_STEP);
}
The code for the `createNewCircle` function is largely the same as the code for `createRandomCircle` in Figure 5-4. The only differences are

1. The `createNewCircle` function creates circles whose initial size is 0.
2. The `createNewCircle` function records the eventual and current size of the circle in the variables `desiredSize` and `currentSize`. Setting both of these variables to 0 ensures that `createNewCircle` is called on the first time step.

The code for the `step` function follows the pseudocode outline on page 198. The only new feature is the call to the `setBounds` method, which resets the location and size of the current circle so that it grows by one pixel in each time step.

### 6.6 Additional GObject subclasses

Ever since Chapter 2, you’ve been using classes from the Stanford Graphics Library to create simple drawings on the screen. So far, however, you have seen only a small part of what the graphics library has to offer. Now that you know how to write programs that involve animation and interactivity, it makes sense to learn more about the graphics library and how to use it. This section introduces three new GObject subclasses—`GArc`, `GPolygon`, and `GCompound`—that allow you to create much more interesting graphical displays.

**The GArc class**

The `GArc` class is used to display an arc formed by selecting part of the perimeter of an oval. The `GArc` function itself takes the following parameters:

```
GArc(x, y, width, height, start, sweep)
```

The first four parameters specify the location and size of the rectangle that encloses the arc and therefore have precisely the same interpretation as those parameters in calls to `GRect` or `GOval`. The next two parameters specify the `start angle`, which is the angle at which the arc begins, and the `sweep angle`, which is the number of degrees through which the arc extends. In keeping with mathematical convention, angles in the graphics library are measured in degrees counterclockwise from the +x axis, as follows:
The geometric interpretation of these parameters is illustrated in Figure 6-8. The effect of these parameters, however, is more easily demonstrated by example. The four sample runs in Figure 6-9 show the effect of the code that appears below each diagram. The code fragments create and display arcs using different values for \texttt{start} and \texttt{sweep}. Each of the arcs has a radius of \( r \) pixels and is centered at the
point \((cx, cy)\). The last two examples show that the values of \texttt{start} and \texttt{sweep} can be negative, in which case the angles extend in the clockwise direction.

The \texttt{GArc} class implements the methods shown in Figure 6-10. As you can see, these methods include \texttt{setFilled} and \texttt{setFillColor}, just as \texttt{GRect} and \texttt{GOval} do. It is not immediately apparent, however, exactly what filling an arc means. The interpretation of arc-filling used in the Stanford Graphics Library seems a bit confusing at first, primarily because the unfilled version of a \texttt{GArc} is not simply the boundary of its filled counterpart. If you display an unfilled \texttt{GArc}, only the arc itself is shown. If you call \texttt{setFilled(true)} on that arc, the graphics library connects the end points of the arc to the center from which the arc was drawn and then fills the interior of that region. The following sample run illustrates the difference by showing both unfilled and filled version of the same 60-degree arc:

![UnfilledAndFilledArcs](image)

The \texttt{UnfilledAndFilledArcs.js} program that produces this output is available on the web site supplied with this book.

The important lesson to take from this example is that the geometric boundary of a \texttt{GArc} changes if you set it to be filled. A filled arc is a wedge-shaped region that has a well-defined interior. An unfilled arc is simply a section taken from the boundary of an ellipse. If you want to display the outline of the wedge that calling \texttt{setFilled} would generate, the easiest way is to call \texttt{setFilled(true)} and then use \texttt{setFillColor("White")} to set the interior of the region to white.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{arc.setFilled(flag)}</td>
<td>Sets whether the wedge for this arc is filled.</td>
</tr>
<tr>
<td>\texttt{arc.setFillColor(color)}</td>
<td>Sets the color used to fill the wedge for this arc.</td>
</tr>
<tr>
<td>\texttt{arc.setStartAngle(start)}</td>
<td>Sets the start angle to \texttt{start}.</td>
</tr>
<tr>
<td>\texttt{arc.getStartAngle()}</td>
<td>Returns the start angle.</td>
</tr>
<tr>
<td>\texttt{arc.setSweepAngle(sweep)}</td>
<td>Sets the start angle to \texttt{sweep}.</td>
</tr>
<tr>
<td>\texttt{arc.getSweepAngle()}</td>
<td>Returns the sweep angle.</td>
</tr>
</tbody>
</table>
The **GPolygon class**

The GPolygon class makes it possible to display a **polygon**, which is simply the mathematical name for a closed shape whose boundary consists of straight lines. The line segments that form the outline of a polygon are called **edges**. The point at which a pair of edges meets is called a **vertex**. Many polygonal shapes are familiar from the real world. Each cell in a honeycomb is a hexagon, which is the common name for a polygon with six sides. A stop sign is an octagon with eight identical sides. Polygons, however, are not required to be so regular. The figures in the right margin, for example, illustrate four polygons that fit the general definition.

The GPolygon class is easy to use if you keep the following points in mind:

- Unlike the functions that create the other shapes, the GPolygon function class does not create the entire figure. What happens instead is that calling GPolygon creates an empty polygon that contains no vertices. Once you have created the empty polygon, you then add vertices to it by calling various other methods described later in this section.

- The origin of a GPolygon is not defined to be its upper left corner. Many polygons, after all, don’t have an upper left corner. What happens instead is that you—as the programmer who is creating the specific polygon—choose a **reference point** that defines the location of the polygon as a whole. You then specify the coordinates for each vertex in terms of where they lie relative to the reference point. The advantage of this design is that you don’t have to recompute the coordinates of each vertex if you move the GPolygon on the graphics window. All you have to do is move the reference point. Because the vertices are defined relative to the reference point, the code that redraws the polygon will shift all the vertices to their correct locations.

The creation of a GPolygon is easiest to illustrate by example. Suppose that you want to create a GPolygon representing the diamond-shaped figure shown in the margin. Your first design decision consists of choosing where to put the reference point. For most polygons, the most convenient point is the geometric center of the figure. If you adopt that model, you then need to create an empty GPolygon and add four vertices to it, specifying the coordinates of each vertex relative to the center. Assuming that the width and height of the diamond are stored in the constants DIAMOND_WIDTH and DIAMOND_HEIGHT, you can create the diamond-shaped GPolygon using the following code:

```javascript
let diamond = GPolygon();
diamond.addVertex(-DIAMOND_WIDTH / 2, 0);
diamond.addVertex(0, DIAMOND_HEIGHT / 2);
diamond.addVertex(DIAMOND_WIDTH / 2, 0);
diamond.addVertex(0, -DIAMOND_HEIGHT / 2);
```
You can then add the diamond at the center of the graphics window by executing the following statement:

```javascript
gw.add(diamond, gw.getWidth() / 2, gw.getHeight() / 2);
```

The graphics window would then look like this:

![DrawDiamond](image)

When you use the `addVertex` method to construct a polygon, the coordinates of each vertex are expressed relative to the reference point. In some cases, it is easier to specify the coordinates of each vertex in terms of the preceding one. To enable this approach, the `GPolygon` class offers an `addEdge` method, which is similar to `addVertex` except that the parameters specify the displacement from the previous vertex to the current one. You can therefore draw the same diamond by making the following sequence of calls:

```javascript
let diamond = GPolygon();
diamond.addVertex(-DIAMOND_WIDTH / 2, 0);
diamond.addEdge(DIAMOND_WIDTH / 2, DIAMOND_HEIGHT / 2);
diamond.addEdge(DIAMOND_WIDTH / 2, -DIAMOND_HEIGHT / 2);
diamond.addEdge(-DIAMOND_WIDTH / 2, -DIAMOND_HEIGHT / 2);
diamond.addEdge(-DIAMOND_WIDTH / 2, DIAMOND_HEIGHT / 2);
```

Note that the first vertex must still be added using `addVertex`, but that subsequent ones can be defined by specifying the edge displacements.

As you work with the `GPolygon` class, you will discover that some polygons are easier to define with successive calls to `addVertex`, while others are easier to define using `addEdge`. For many polygonal figures, however, it is even more convenient to define the edges using polar coordinates. The `GPolygon` class supports this style through the method `addPolarEdge`. This method is identical to `addEdge` except that its arguments are the length of the edge and its direction is expressed in degrees counterclockwise from the +x axis.

The `addPolarEdge` method makes it easy to create figures in which you know the angles of the edges but would need trigonometry to calculate the vertices. The following function, for example, uses `addPolarEdge` to create a regular hexagon in which the length of each edge is determined by the parameter `side`:
function createHexagon(side) {
    let hex = GPolygon();
    hex.addVertex(-side, 0);
    let angle = 60;
    for (let i = 0; i < 6; i++) {
        hex.addPolarEdge(side, angle);
        angle -= 60;
    }
    return hex;
}

As always, the first vertex is added using `addVertex`. Here, the initial vertex is the one at the left edge of the hexagon. The first edge then extends from that point at an angle of 60 degrees. Each subsequent edge has the same length, but sets off at an angle 60 degrees to the right of the preceding one. When all six edges have been added, the final edge will end up at the original vertex, thereby closing the polygon.

Once you have defined this method, executing the statement

```javascript
    gw.add(createHexagon(50), gw.getWidth() / 2, gw.getHeight() / 2);
```

would produce the following display:

![Hexagon](image)

Figure 6-11 lists the methods that apply to the `GPolygon` class. As with the other bounded figures, `GPolygon` implements `setFilled` and `setFillColor`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>poly.addVertex(x, y)</code></td>
<td>Adds a vertex at the point (x, y).</td>
</tr>
<tr>
<td><code>poly.addEdge(dx, dy)</code></td>
<td>Adds a vertex shifted by dx and dy from the previous one.</td>
</tr>
<tr>
<td><code>poly.addPolarEdge(r, theta)</code></td>
<td>Adds a vertex shifted by r units in direction theta.</td>
</tr>
<tr>
<td><code>poly.setFilled(flag)</code></td>
<td>Sets whether the polygon is filled.</td>
</tr>
<tr>
<td><code>poly.setFillColor(color)</code></td>
<td>Sets the color used to fill the polygon.</td>
</tr>
</tbody>
</table>
As another example of using the `GPolygon` class, the `createStar` function in Figure 6-12 creates a `GPolygon` whose edges form a five-pointed star, as shown in the following display:

![Star Diagram]

Although the star is more complicated mathematically than the earlier examples, the most difficult part is determining the coordinates of the starting point at the left edge of the star. Calculating the $x$ coordinate is easy because the starting point is simply half the width of the star to the left of its center. Calculating the distance in the $y$ direction requires a bit of trigonometry, which can be illustrated as follows:

![Trigonometry Diagram]

```javascript
/*
 * Creates a GPolygon representing a five-pointed star with the reference
 * point at the center. The size refers to the width of the star at its
 * widest point.
 */

function createStar(size) {
    let poly = GPolygon();
    let dx = size / 2;
    let dy = dx * Math.tan(18 * Math.PI / 180);
    let edge = dx - dy * Math.tan(36 * Math.PI / 180);
    poly.addVertex(-dx, -dy);
    let angle = 0;
    for (let i = 0; i < 5; i++) {
        poly.addPolarEdge(edge, angle);
        poly.addPolarEdge(edge, angle + 72);
        angle += 72;
    }
    return poly;
}
```
Each of the points around the periphery of a five-pointed star forms an angle that is a tenth of a complete circle, which is 36 degrees. If you draw a line that bisects that angle—leaving 18 degrees on either side—that line will hit the geometric center of the star, forming the right triangle shown in the diagram. The value of \( dy \) is therefore equal to \( dx \) multiplied by the tangent of 18 degrees. The other tricky calculation is that of the edge length, which is illustrated in the following diagram:

![Diagram showing a right triangle with edge, dx, and dy labeled, forming a five-pointed star with a dotted line bisecting one of the angles.]  

To determine the value of \textit{edge}, you need to subtract the dotted portion of the horizontal line from its entire length, which is given by \( dx \). The length of the dotted portion is easily computed using trigonometry as \( dy \) multiplied by the tangent of 36 degrees. Once you have computed these values, the rest of the \textit{createStar} function follows much the same pattern as the implementation of \textit{createHexagon}.

**The GCompound class**

The \textit{GCompound} class makes it easy to assemble a collection of graphical objects into a single unit. As with \textit{GPolygon}, calling \textit{GCompound} creates an empty structure that you then have to fill by calling \textit{add}, just as if you were adding those objects to the graphics window. Once you have added the objects, you can then add the whole \textit{GCompound} to the window, at which point it functions as a single object.

Like the \textit{GPolygon} class, the \textit{GCompound} class defines its own coordinate system in which all coordinate values are expressed relative to a reference point. This design makes it possible for you to define a \textit{GCompound} without having to know exactly where it is going to appear.

As a simple example, the function \textit{createCrossedBox} that appears at the bottom of Figure 6-13 creates a \textit{GCompound} consisting of a rectangle and its two diagonal lines, which appears like this in the graphics window:
/*
 * File: RotateCrossedBox.js
 * ------------------------
 * This program draws a crossed box and then rotates it around its center.
 */

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 200;
const BOX_WIDTH = 200;
const BOX_HEIGHT = 100;
const TIME_STEP = 20;
const N_STEPS = 360;

/*
 * Draws a crossed box and then rotates it around its center.
 */

function RotateCrossedBox() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let box = createCrossedBox(BOX_WIDTH, BOX_HEIGHT);
    gw.add(box, gw.getWidth() / 2, gw.getHeight() / 2);
    let stepCount = 0;

    function step() {
        if (stepCount < N_STEPS) {
            box.rotate(1);
            stepCount++;
        } else {
            clearInterval(timer);
        }
    }

    let timer = setInterval(step, TIME_STEP);
}

/*
 * Creates a crossed box, which is a compound consisting of a GRect and
 * its two diagonals. The reference point is at the center of the figure.
 */

function createCrossedBox(width, height) {
    let compound = GCompound();
    compound.add(GRect(-width / 2, -height / 2, width, height));
    compound.add(GLine(-width / 2, -height / 2, width / 2, height / 2));
    compound.add(GLine(-width / 2, height / 2, width / 2, -height / 2));
    return compound;
}
Summary

In this chapter, you learned how to create interactive programs. The important points introduced in this chapter include:

- Interactive programs in JavaScript use an event-driven model in which user actions generate events that occur asynchronously with respect to the operation of the program. Each event triggers a function call that responds to that event.
- Functions in JavaScript are first-class values in the sense that they can be used in all the ways that any other value can. Functions can be assigned to variables, passed as parameters to other functions, and returned as a function result.
- The GObject class exports a large collection of methods that apply to every graphical object. A list of these methods appears in Figure 6-2 on page 188.
- Programs indicate their interest in responding to mouse events by calling the `addEventListener` method on the graphics window.
- Mouse events are associated with an event type indicated by a string. The names of the different event types appear in Figure 6-3 on page 189.
- Each call to `addEventListener` specifies the function that should respond to that type of event. These functions are generically known as callback functions.
- Callback functions are conventionally declared within the body of an enclosing function so that the callback function has access to the local variables of the function in which the callback function is declared.
- Callback functions used to respond to mouse events take a single parameter that includes information about the event. The only mouse-event properties used in this text are the methods `getX` and `getY`, which return the position in the window at which the mouse event occurred.
- The GWindow class includes a method `getElementAt(x, y)` that returns the GObject at that location in the window. If there is no object at that location, `getElementAt` returns the special value `null`.
- The usual strategy for implementing animation in a JavaScript program is to use a `timer`, which executes a callback function after a specified delay. If the delay is 20 milliseconds or less, motion on the screen appears continuous to the eye.
- JavaScript provides two kinds of timers. The function `setTimeout` creates a one-shot timer that triggers an event after a delay. The function `setInterval` creates an interval timer that triggers an event repeatedly every time the delay time expires.
- Local variables in JavaScript are defined only in the context of the block in which they are declared. The set of names accessible inside a block is called its scope.
• The GArc class makes it possible to display elliptical arcs defined by a bounding rectangle and two angles: a start angle that indicates where the arc starts and a sweep angle that indicates how far the arc extends. Filled arcs appear as wedges in which the endpoints of the arc are connected to the center.

• The GPolygon class makes it possible to display an arbitrary polygon. The GPolygon function itself creates an empty polygon; you create the actual polygon by calling some combination of the methods addVertex, addEdge, and addPolarEdge.

• The GCompound class is a GObject subclass that contains other GObject instances and allows the collection to be treated as a unit.

Review questions

1. In the context of JavaScript, what is an event?

2. Are events in JavaScript synchronous or asynchronous?

3. What reasons are offered in this chapter for the close association of graphical user interfaces and object-oriented programming?

4. Why are functions in JavaScript said to be first-class functions?

5. True or false: In JavaScript, you can use an explicit function definition as a parameter to some other function.

6. What are the two parameters to the addEventListener method?

7. What event type do you use to respond to a mouse click?

8. What are the two methods used in this chapter to get more specific information about a mouse event?

9. What is a callback function?

10. How does a callback function typically share information with the function that defines it?

11. What is meant by the term rubber-banding?

12. What value does the getElementAt method return if no object exists at the specified location?

13. How does the getElementAt method decide which object to return if more than one object covers the specified location?
14. Describe in your own words the strategy for implementing animation in JavaScript?

15. What is a **timer**?

16. What are the two library functions that create timers? How do they differ?

17. How do you stop a timer?

18. True or false: Functions declared as part of a block have access to the local variables declared earlier in the block, but not to those declared later.

19. What is meant by the term **scope**?

20. Describe the significance of the *start* and *sweep* parameters in the call to the **GArc** function.

21. What does it mean if the *sweep* argument to the **GArc** constructor is negative?

22. Describe the arcs produced by each of the following calls to the **GArc** function, where *cx* and *cy* are the coordinates of the center of the window and *r* has the value 100:

   a) **GArc(cx, cy, 2 * r, 2 * r, 0, 270)**
   b) **GArc(cx, cy, 2 * r, 2 * r, 135, -90)**
   c) **GArc(cx, cy, 2 * r, 2 * r, 180, -45)**
   d) **GArc(cx, cy, 3 * r, r, -90, 180)**

23. How does the **GArc** class interpret the notion of a filled arc?

24. Describe the differences between the **GPolygon** methods **addVertex**, **addEdge**, and **addPolarEdge**.

25. True or false: The **addVertex** method is always used to add the first vertex to a **GPolygon**.

26. In your own words, describe the purpose of the **GCompound** class.

---

**Exercises**

1. Drawing on the **printFunctionTable** function for inspiration, implement a function

   ```javascript
   function plot(gw, f, xmin, xmax, ymin, ymax)
   ```

   that plots the function *f* on the graphics window by creating small **GLine** segments and adding them to the graphics window. The parameters *xmin*,
**xMax**, **yMin**, and **yMax** specify a translation between the values passed to and returned by the function and the window coordinates. The left edge of the window, for example, should correspond to the value **xMin** in the domain of the function.

For example, calling

```javascript
plot(gw, Math.sin, -2 * Math.PI, 2 * Math.PI, -1, 1);
```

should generate a plot of the trigonometric sine function for values of \( x \) ranging from \(-2\pi\) to \(+2\pi\) and displayed so that the vertical space in the window runs from \(-1\) at the bottom to \(+1\) at the top (note that this interpretation requires you to flip JavaScript’s coordinate system so that it matches the traditional Cartesian model in which \( y \) values increase as you move upward). The graphics window after making this call should therefore look like this:

![Plot](image)

Similarly, calling

```javascript
plot(gw, Math.sqrt, 0, 4, 0, 2);
```

should plot the **Math.sqrt** function on a graph that extends from 0 to 4 along the \( x \)-axis and from 0 to 2 along the \( y \)-axis, like this:

![Plot](image)

2. Modify the **DrawDots** program so that clicking the mouse draws a small \( \times \) every time you click the mouse. The \( \times \), which consists of two **GLine** objects, should be positioned so that the intersection appears at the point at which the mouse was clicked.
3. In addition to line drawings of the sort generated by the DrawLines program, interactive drawing programs allow you to add other shapes to the canvas. In a typical drawing application, you create a rectangle by pressing the mouse at one corner and then dragging it to the opposite corner. For example, if you press the mouse at the location in the left diagram and then drag it to the position in which you see it in the right diagram, the program creates the rectangle shown:

The rectangle grows as you drag the mouse. When you release the mouse button, the rectangle is complete and stays where it is. You can then go back and add more rectangles in the same way.

Although the code for this exercise is quite short, there is one important consideration that you will need to take into account. In the example above, the initial mouse click is in the upper left corner of the rectangle. Your program, however, has to work just as well if you drag the mouse in some other direction besides to the right and down. For example, you should also be able to draw a rectangle by dragging to the left, as shown in the following illustration:

4. Use the GOval, GLine, and GRect classes to create a cartoon drawing of a face that looks like this:
Once you have this picture, add a callback function for the "mousemove" event so that the pupils in the eyes follow the cursor position. For example, if you move the cursor to the lower right side of the screen, the pupils should shift so that they appear to be looking at that point, as follows:

![DrawFace](image)

Although it doesn’t matter much when the cursor is outside the face, it is important to compute the position of the pupil independently for each eye. If you move the mouse between the eyes, for example, the pupils should point in opposite directions so that the face appears cross-eyed.

5. Write a program that draws a filled black square in the center of the canvas. Once you have that part of the program working, animate your program so that the color of the square changes once a second to a new color chosen randomly by calling the `randomColor` function in the `RandomLib.js` library. Your program should run for a minute and then stop.

6. Using the `AnimatedSquare.js` program as a model, write a program that bounces a ball inside the boundaries of the graphics window. Your program should begin by placing a `GOval` in the center of the window to represent the ball. On each time step, your program should shift the position of the ball by `dx` and `dy` pixels, where both `dx` and `dy` initially have the value 1. Whenever the leading edge of the ball touches one of the boundaries of the window, your program should make the ball bounce by negating the value of `dx` or `dy`, as appropriate. Don’t worry about getting your program to stop.

7. Rewrite the program from exercise 6 so that the ball is implemented as a `GCompound` containing a `GOval` shifted by the radius of the ball in both the `x` and `y` directions. The advantage of making this change is that the coordinates of the `GCompound` now refer to the center of the ball, which makes the code to see whether the ball is bouncing more symmetrical and therefore easier to understand.
8. Write a program that draws a picture of a pumpkin pie divided into equal wedge-shaped pieces where the number of pieces is indicated by the constant \textit{N\_PIECES}. Each wedge should be a separate \texttt{G\_Arc}, filled in orange and outlined in black. The following screen image, for example, shows the diagram when \texttt{N\_PIECES} is 6.

![Pumpkin Pie Diagram](image1)

Once you have this display, add event processing to your application so that clicking on any of the wedges removes that wedge from the display. For example, if you click on the wedge in the upper right, the screen image should look like this:

![Pumpkin Pie Diagram](image2)

9. The title character in the PacMan series of games is easy to draw in Java using a filled \texttt{G\_Arc}. As a first step, write a program that adds a PacMan figure at the left edge of the window, as follows:

![PacMan Image](image3)

Once you have this part working, add the code to make the PacMan figure move rightward until it reaches the right edge of the graphics window. As
PacMan moves, your program should change the start and sweep angles so that the mouth appears to open and close as illustrated in the following sequence of images:

10. In J. K. Rowling’s *Harry Potter and the Deathly Hallows*, those who believe in the legend named in the title recognize one another through a symbol that combines three elements—a triangle representing the cloak of invisibility, a circle representing the stone of resurrection, and a line representing the elder wand—superimposed as follows:

Write a function `createDeathlyHallowsSymbol` that takes two parameters indicating the width and height of the figure and returns a `GCompound` that includes all three of these elements. The triangle should be a `GPolygon`, the circle should be a `GOval`, and the line should be a `GLine`. The geometry is straightforward for both the line and the triangle, but rather complicated for the circle, which must exactly touch the edges of the triangle. Although you can figure out the necessary relationships by using the Pythagorean theorem, we’ve already done that calculation, and you can simply use the following formula for the radius $r$ as a function of the width $w$ and the height $h$:

$$ r = \frac{w \sqrt{h^2 + \frac{w^2}{4} - \frac{w^2}{2}}}{2h} $$

Use the `createDeathlyHallowsSymbol` function to write a program that displays the symbol in the center of the window. Once you’ve done that, add the code needed to let the user drag the symbol around the window.

11. Write a program to play the classic arcade game of Breakout, which was developed in 1976 by Steve Wozniak, who would later become one of the founders of Apple. In Breakout, your goal is to clear a collection of bricks by hitting each of them with a bouncing ball.
The initial configuration of the Breakout game appears in the leftmost diagram in Figure 6-14. The colored rectangles in the top part of the screen are bricks, two rows each of red, orange, yellow, green, and blue. The slightly larger rectangle at the bottom is the paddle. The paddle is in a fixed position in the vertical dimension, but moves back and forth across the screen along with the mouse until it reaches the edge of its space.

A complete Breakout game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world. Thus, after two bounces—one off the paddle and one off the right wall—the ball might have the trajectory shown in the middle diagram.

As you can see from the middle diagram, the ball is about to collide with one of the bricks on the bottom row. When that happens, the ball bounces just as it does on any other collision, but the brick disappears (which you can accomplish simply by removing it from the graphics window). The play continues in this way until one of the following conditions occurs:

- The ball hits the lower wall, which means that you must have missed it with the paddle. In this case, the turn ends and the next ball is served, assuming that you have not already exhausted your allotment of three turns. If you have, the game ends in a loss.
- The last brick is eliminated. In this case, the game ends immediately, and you can retire victorious.

After all the bricks in a particular column have been cleared, a path will open to the top wall, as shown in the rightmost diagram in Figure 6-14. When
this delightful situation occurs, the ball will often bounce back and forth several times between the top wall and the upper line of bricks without the user ever having to worry about hitting the ball with the paddle. This condition is called “breaking out.” It is important to note that, even though breaking out is a very exciting part of the player’s experience, you don’t have to do anything special in your program to make it happen. The game operates the same as always: balls bounce off walls, collide with bricks, and obey the laws of physics.

The only part of the implementation that requires some explanation is the problem of checking to see whether the ball has collided with a brick or the paddle. The `getElementAt` method can determine whether there is an object at a particular position, but it doesn’t work well to check the coordinates of the center because the ball is larger than a single point. In this program, the simplest strategy is to check the four corner points on the square in which the ball is inscribed. A collision occurs if any of those points are inside a brick.

12. In New York’s Times Square, you can get the news of the day by watching headlines on large display screens that show a single line of text. The headline initially begins to appear at the right edge of the screen and then moves quickly from right to left. Your job in this exercise is to write a program that simulates this type of headline display by moving a `GLabel` across the screen.

Suppose, for example, that you want to use your program to display the famous `Chicago Tribune` headline from when the paper incorrectly called the result of the 1948 presidential election:

```
DEWEY DEFEATS TRUMAN
```

Your program should create a `GLabel` containing the headline and then position it so that the entire text of the label is clipped beyond the right edge of the screen. Your program should then implement a timer-based graphical animation that moves the `GLabel` a few pixels to the left on each time step. After a few seconds, the entire first word will be visible:

```
DEWEY
```

As the label continues to scroll, letters will disappear off the left edge of the screen as new letters appear on the right. Your program should continue to scroll letters toward the left until the entire `GLabel` disappears from view.
The work [of conducting the census should] be done so far as possible by mechanical means. In order to accomplish this the records must be put in such shape that a machine could read them. This is most readily done by punching holes in cards.

— Herman Hollerith, *An Electric Tabulating System*, 1889

The idea of encoding text in machine-readable form dates back to the nineteenth century and the work of the American inventor Herman Hollerith. After studying engineering at City College of New York and the Columbia School of Mines, Hollerith spent a couple of years working as a statistician for the U.S. Census Bureau before accepting a teaching position at MIT. While at the Census Bureau, Hollerith had become convinced that the data produced by the census could be counted more quickly and accurately by machine. In the late 1880s, he designed and built a tabulating machine that was used to conduct the 1890 census in record time. The company he founded to commercialize his invention, originally called the Tabulating Machine Company, changed its name in 1924 to International Business Machines (IBM). Hollerith’s card-based tabulating system pioneered the technique of textual encoding described in this chapter—a contribution that was reflected in the fact that early versions of the FORTRAN language used the letter H (for Hollerith) to indicate text data.
Strings

Although you have been using strings ever since Chapter 2, you have only scratched the surface of what you can do with string data. This chapter introduces the features available in the JavaScript String class, which provides a convenient abstraction for working with strings of characters. Understanding the various methods available in this library will make it much easier to write interesting applications. Before looking at the details of the String class, however, it helps to take a step back and look at how computers store data in the first place.

7.1 Binary representation

Today’s computers represent information in a simple but powerful form that allows any information—no matter how complex—to be stored as a sequence of primitive values, each of which can be in only one of two possible states. Each of those primitive values is called a bit.

The interpretation of the values for each bit depends on how you choose to view the underlying information. If you think of the bits that form the internal circuitry of the machine as tiny light switches, you might label those states as off and on. If you think of each bit as a logical value, you might instead use the Boolean labels false and true. However, because the word bit comes originally from a contraction of binary digit, it is more common to label those states as 0 and 1, which are the digits of the binary number system on which computer arithmetic is based.

Binary notation

The idea of writing numbers in binary notation predates the development of the electronic computer by more than 250 years. The German mathematician Gottfried Wilhelm von Leibniz (1646–1716) offered a detailed account of the binary system in a paper published by the French Royal Academy of Science in 1703. In that paper, Leibniz writes:

Ordinary arithmetic calculation is performed following a progression by tens. One uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, which signify zero, one, and the following numbers up to nine, inclusive. On going up to ten, one starts again, and writes ten as 10; ten times ten, or one hundred, as 100; ten times one hundred, or one thousand, as 1000; and ten times a thousand as 10000. And so on.

But instead of the progression by tens, I have used for several years the simplest progression of all, which goes by twos, which I find to be the perfection of the science of numbers. I therefore do not use any characters other than 0 and 1, and on going up to two, I start again. That is why two is written here as 10; and two times two or four as 100; and two times four or eight as 1000 . . .

As Leibniz’s description makes clear, it is easy to translate a number written in binary back to its decimal equivalent. All you need to do is add together the place
values of each digit in the number, keeping in mind that each digit in a binary number counts for twice as much as its neighbor on the right. For example, if Leibniz were to use binary notation to represent the year of his birth, he would write the number down like this:

\[ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \]

The following diagram shows that this value indeed corresponds to the value 1646:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
\times & 1 & = & 0 \\
\times & 2 & = & 2 \\
\times & 4 & = & 4 \\
\times & 8 & = & 8 \\
\times & 16 & = & 0 \\
\times & 32 & = & 32 \\
\times & 64 & = & 64 \\
\times & 128 & = & 0 \\
\times & 256 & = & 0 \\
\times & 512 & = & 512 \\
\times & 1024 & = & 1024 \\
\end{array}
\]

\[ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \]

For the most part, numeric representations in this book use decimal notation for readability. If the base is not clear from the context, the text follows the usual strategy of using a subscript to denote the base. For example, the equivalence of the binary value 11001101110 and the decimal value 1646 can be made explicit by writing the numbers like this:

\[ 11001101110_2 = 1646_{10} \]

**Storing integers as sequences of bits**

The binary representation described by Leibniz makes it easy to store integers as a sequence of individual bits. In modern computer hardware, individual bits are collected together into larger units that are then treated as integral units of storage. The smallest such combined unit is called a *byte*, which consists of eight bits. Bytes are then assembled into larger structures called *words*, where a word is usually defined to be the size required to hold an integer value of the type most appropriate for the hardware. Today, machines typically organize their memory into words that are either four or eight bytes long (32 or 64 bits).

To get a sense of how computers can store integers internally, consider the byte containing the following binary digits:

\[ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \]
That sequence of bits represents the number forty-two, which you can verify—just as Leibniz would have done—by calculating the contribution for each of the individual bits, as follows:

Bytes can store integers between 0 and 255, which is $2^8 - 1$. Numbers outside this range must be stored in larger units that use more bits of memory.

**Hexadecimal notation**

Although the bit diagrams make it clear how computers store integer values internally, these diagrams also demonstrate the fact that writing numbers in binary form is terribly inconvenient. Binary numbers are cumbersome, mostly because they tend to be so long. Decimal representations are intuitive and familiar but make it harder to understand how the number translates into bits.

For applications in which it is useful to understand how a number translates into its binary representation without having to work with binary numbers that stretch all the way across the page, computer scientists tend to use *hexadecimal* (base 16) notation instead. In hexadecimal notation, there are sixteen digits that represent the values from 0 to 15. Although the decimal digits 0 through 9 are perfectly adequate for the first ten digits, classical arithmetic does not define the extra symbols you need to represent the remaining six. Computer science traditionally uses the letters A through F for this purpose, as follows:

- $A = 10$
- $B = 11$
- $C = 12$
- $D = 13$
- $E = 14$
- $F = 15$

What makes hexadecimal notation useful is the fact that you can easily convert between hexadecimal values and the underlying binary representation. All you need to do is combine the bits into groups of four. For example, the number forty-two can be converted from binary to hexadecimal like this:
The first four bits represent the number 2, and the next four represent the number 10. Converting each of these to the corresponding hexadecimal digit gives \texttt{2A} as the hexadecimal form. You can then verify that this number still has the value 42 by adding up the digit values, as follows:

\[
\begin{array}{c}
2 \\
A \\
\times 1 = 10 \\
\times 16 = 32 \\
\hline
42
\end{array}
\]

For the most part, numeric representations in this book use decimal notation for readability. As noted earlier, the text follows the convention of using a subscript to denote the base if it is not clear from context. Thus, the three most common representations for the number forty-two—decimal, binary, and hexadecimal—look like this:

\[
42_{10} = 00101010_2 = 2A_{16}
\]

The most important thing to remember, however, is that the number itself is always the same; the numeric base affects only the representation. Forty-two has an intrinsic meaning that is independent of the base, which is perhaps easiest to see in the representation an elementary school student might use:

The number of tick marks in this representation is forty-two. The fact that a number is written in binary, decimal, or any other base is a property of the representation, not of the number itself. Numbers do not have bases; representations do.

**Base conversion using parseInt and toString**

JavaScript makes it easy to convert back and forth between numeric values and their string representations in any base. The built-in function \texttt{parseInt}, which converts a string of digits into the corresponding integer, takes an optional second argument specifying the base. For example, calling \texttt{parseInt("2A", 16)} returns the numeric value 42. If the second argument is missing, \texttt{parseInt} uses base 10. The \texttt{parseInt} function and the corresponding \texttt{parseFloat} function for real numbers are described in more detail in section 7.5.
The `toString` method—which can be applied to any JavaScript value—converts that value to its string representation. For numbers, `toString` allows you to specify the base as a parameter, so that, for example, if `n` contains the value 42, calling `n.toString(16)` returns the string "2a". If you prefer upper case for hexadecimal digits, you can call `toUpperCase`, as described later in this chapter.

**Representing nonnumeric data**

The preceding sections in this chapter have focused on how computers store numbers. This chapter, however, is supposed to be about strings, which are an important example of nonnumeric data. The challenge in having computers represent nonnumeric data lies in finding a way to represent that information inside the computer.

The simplest strategy for representing nonnumeric data is to assign numbers to the individual data values you need to represent. For example, the conventional way to represent the months of the year—even without a computer—is to give each month a number: January has the value 1, February has the value 2, and so on, up to December, which has the value 12. This strategy is called enumeration.

Once you have enumerated a set of values, you can represent those values in memory by using the appropriate numeric code. For example, the numeric value 12 that corresponds to the month of December. Internally, that value is stored as an integer expressed—as you saw in the preceding sections—as a sequence of binary digits. There is no indication in the hardware as to whether that value represents the integer 12 or the numeric representation for the month of December. The meaning of a particular value depends on how it is used. If the program uses the value arithmetically, it is interpreted as the integer 12. If it instead uses that value to select from a list of month names, that value indicates December. In either case, the number stored inside the computer is exactly the same.

The strategy of using numbers to represent nonnumeric data is one of the most important ideas in the history of computation. One of the clearest and earliest expositions of that idea comes from Ada Lovelace, daughter of the poet Lord Byron and his wife Anna Isabella Byron. In the 1840s, Lady Lovelace collaborated with the English mathematician and inventor Charles Babbage on the design of his Analytical Engine, which anticipated several essential features of modern computers including the ability to solve different tasks by changing its programming. Indeed, much of what we know about the Analytical Engine—which sadly was never completed—comes from Lovelace’s translation of a detailed description of Babbage’s work by the Italian engineer Luigi Menabrea. Her translation, entitled *Sketch of the Analytical Engine Invented by Charles Babbage, Esq.*, was published in 1843, along with her explanatory notes that were almost three times as long as the original paper. Lovelace recognized that the algebraic patterns for which the
Analytical Engine was designed could be extended to include concepts beyond simple numbers, expressing her conviction that “the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.”

In an interview for a film about Ada Lovelace’s life and work, Doron Swade, who led the effort to rebuild Babbage’s Difference Engine for the Science Museum in London, offers the following description of Ada’s contribution:

Ada saw something that, in some sense, Babbage failed to see. In Babbage’s world, his engines were bound by number. . . . What Lovelace saw—what Ada Byron saw—was that number could represent entities other than quantity. So, once you had a machine for manipulating numbers, if those numbers represented other things—letters, musical notes—then the machine could manipulate symbols of which number was one instance.”

**Representing characters**

The primitive elements of string data are individual characters. Like the months of the year, characters can be represented inside the computer by assigning each character a numeric code. You could, for example, assign successive integers to represent each of the letters in the alphabet, using 0 for the letter A, 1 for letter B, and so on. In 1605, the English philosopher and scientist Francis Bacon did precisely that when he devised a technique for encoding messages that is now known as Bacon’s cipher. What is, however, even more astonishing is that Bacon based his cipher on the binary representation of these numbers, almost a century before Leibniz published his paper on binary arithmetic. Bacon’s cipher, however, was not used in practice and had little or no influence on the later development of computation.

The first binary encoding scheme for characters used extensively in practice was the Baudot code, which was invented in 1870 by the French engineer Émile Baudot, one of the pioneers of the telegraph. In Baudot’s scheme, each of the 26 letters was assigned a numeric code. The encoding also included a few special characters to represent the space character, the two characters telegraph printers used to designate the end of a line, and transitions to an alternate character set used for digits and punctuation. The letters of the alphabet did not appear in order, but were instead chosen so that the most common letters, such as E and T, would require pressing just one of the five keys on the input device.

The fact that the letters do not appear consecutively in the Baudot code does not make it any less effective as an encoding scheme. The only essential characteristic of an encoding scheme is that the sender and receiver agree on how to convert letters to numeric codes. The need for a common encoding shared by senders and receivers increases the importance of standardization. As long as all telegraph operators used the same code, they were able to communicate with one another.
In the early years of the computing industry, standardization was complicated by the existence of two incompatible character encodings. The American Standards Association (now known as the American National Standards Institute or ANSI) began work on a standardized character encoding in 1960, which was formalized in 1963 as the *American Standard Code for Information Interchange* or *ASCII*. Early IBM machines, however, used a different character set derived from the coding system used for devices that worked with punched cards. When it released the System/360 operating system in 1964, IBM chose to create a competing coding scheme called the *Extended Binary Coded Decimal Interchange Code* or *EBCDIC*. The two encoding schemes coexisted for several decades, but ASCII and its successors have become nearly universal in recent years.

In its original design, the ASCII character set contained 128 characters, which is enough to store the uppercase and lowercase letters of the Roman alphabet, the standard decimal digits, a variety of punctuation symbols, and a set of nonprinting characters called *control characters*. The characters in the ASCII set appear in Figure 7-1. The control characters are indicated in the table using a backslash (\), followed either by a single-character abbreviation for the control character or the numeric value of the character expressed in base 8. Control characters will be introduced in this text only on the rare occasions that you need to know them.

The characters are arranged in Figure 7-1 according to their internal values, which are expressed here in hexadecimal. The character A, for example, appears in the row labeled 4x and the column labeled 1, so its internal representation is 41_{16}, which is equal to the decimal number 65. The assignment of internal values to characters is in some sense arbitrary, but you can rely on the following properties:

- The digit characters are consecutive.
- The uppercase and lowercase letters form two consecutive sequences.

**Figure 7-1** The ASCII character set, which matches the first 128 characters of Unicode

<table>
<thead>
<tr>
<th></th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
<th>0x04</th>
<th>0x05</th>
<th>0x06</th>
<th>0x07</th>
<th>0x08</th>
<th>0x09</th>
<th>0x0A</th>
<th>0x0B</th>
<th>0x0C</th>
<th>0x0D</th>
<th>0x0E</th>
<th>0x0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\00</td>
<td>\01</td>
<td>\02</td>
<td>\03</td>
<td>\04</td>
<td>\05</td>
<td>\06</td>
<td>\07</td>
<td>\08</td>
<td>\09</td>
<td>\0A</td>
<td>\0B</td>
<td>\0C</td>
<td>\0D</td>
<td>\0E</td>
<td>\0F</td>
</tr>
<tr>
<td>1</td>
<td>\020</td>
<td>\021</td>
<td>\022</td>
<td>\023</td>
<td>\024</td>
<td>\025</td>
<td>\026</td>
<td>\027</td>
<td>\028</td>
<td>\029</td>
<td>\02A</td>
<td>\02B</td>
<td>\02C</td>
<td>\02D</td>
<td>\02E</td>
<td>\02F</td>
</tr>
<tr>
<td>2</td>
<td>\030</td>
<td>\031</td>
<td>\032</td>
<td>\033</td>
<td>\034</td>
<td>\035</td>
<td>\036</td>
<td>\037</td>
<td>\038</td>
<td>\039</td>
<td>\03A</td>
<td>\03B</td>
<td>\03C</td>
<td>\03D</td>
<td>\03E</td>
<td>\03F</td>
</tr>
<tr>
<td>3</td>
<td>\040</td>
<td>\041</td>
<td>\042</td>
<td>\043</td>
<td>\044</td>
<td>\045</td>
<td>\046</td>
<td>\047</td>
<td>\048</td>
<td>\049</td>
<td>\04A</td>
<td>\04B</td>
<td>\04C</td>
<td>\04D</td>
<td>\04E</td>
<td>\04F</td>
</tr>
<tr>
<td>4</td>
<td>\050</td>
<td>\051</td>
<td>\052</td>
<td>\053</td>
<td>\054</td>
<td>\055</td>
<td>\056</td>
<td>\057</td>
<td>\058</td>
<td>\059</td>
<td>\05A</td>
<td>\05B</td>
<td>\05C</td>
<td>\05D</td>
<td>\05E</td>
<td>\05F</td>
</tr>
<tr>
<td>5</td>
<td>\060</td>
<td>\061</td>
<td>\062</td>
<td>\063</td>
<td>\064</td>
<td>\065</td>
<td>\066</td>
<td>\067</td>
<td>\068</td>
<td>\069</td>
<td>\06A</td>
<td>\06B</td>
<td>\06C</td>
<td>\06D</td>
<td>\06E</td>
<td>\06F</td>
</tr>
<tr>
<td>6</td>
<td>\070</td>
<td>\071</td>
<td>\072</td>
<td>\073</td>
<td>\074</td>
<td>\075</td>
<td>\076</td>
<td>\077</td>
<td>\078</td>
<td>\079</td>
<td>\07A</td>
<td>\07B</td>
<td>\07C</td>
<td>\07D</td>
<td>\07E</td>
<td>\07F</td>
</tr>
<tr>
<td>7</td>
<td>p q r s t u v w x y z { } ^ _</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The ASCII coding system, however, was developed in the United States and was in many ways specific to that environment. For example, the character set includes the currency symbol for the U.S. dollar, but omits symbols like £, €, and ¥ used for other major currencies. ASCII is, moreover, closely tied to the Roman alphabet. Particularly with the rise of the World Wide Web in the 1990s, it was necessary to expand the encoding system to embrace a broader collection of languages. The result of that expansion was a new standard called Unicode, which is used in JavaScript to ensure that it is more universal in its application. The original versions of Unicode allowed for 65,536 ($2^{16}$) characters, but even that number proved to be insufficient. The current Unicode standard allows for 1,114,112 characters, which allows it to represent not only the languages that are in use today, but also the languages of antiquity and even alphabets from fantasy, such as J. R. R. Tolkien’s Tengwar script from *Lord of the Rings*.

### 7.2 String operations

In Chapter 2 you learned that you can use the `+` operator to join strings together end to end. In programming contexts, this operation is called *concatenation*. If you apply the `+` operator to two numbers, JavaScript adds them numerically. If the value on either side of the `+` operator is a string, JavaScript converts both values into strings and concatenates the two.

In addition to the `+` operator, JavaScript allows you to use the relational operators `===`, `!==`, `<`, `<=`, `>`, and `>=` to compare two string values. For example, the following code checks whether the value of `str` is equal to "yes":

```javascript
if (str === "yes") . . .
```

The relational operators compare strings using *lexicographic order*, which is similar to traditional alphabetical order but which uses the underlying Unicode values of each character to make the comparison. Lexicographic order means that case is significant, so "a" is not equal to "A". In lexicographic order, "a" is greater than "A" because the Unicode value for a lowercase a (97) is greater than the Unicode value for an uppercase A (65).

All other string operations require calling methods on a string value using the receiver syntax you learned in Chapter 2 for working with graphical objects:

```javascript
receiver.name(arguments)
```

Figure 7-2 on the next page lists the most common methods that JavaScript defines as part of its *String* class. These methods are explored in more detail in the individual sections that follow.
**Figure 7-2** Common operations in JavaScript’s string class

### String operators

- `$str_1 + str_2$`: Concatenates `str_1` and `str_2` end to end and returns a new string containing the combined characters. As either operand is a string, JavaScript will convert the other to its string form.

- `$str += suffix$`: Appends `suffix` to the end of `str`.

- `$str_1 === str_2$, $str_1 !== str_2$`
- `$str_1 < str_2$, $str_1 <= str_2$`
- `$str_1 > str_2$, $str_1 >= str_2$`: These operators compare `str_1` and `str_2`. The comparison is performed using lexicographic order, which is the order defined by the underlying Unicode values.

### String field

- `str.length`: Returns the number of characters in `str`.

### String class method

- `String.fromCharCode(code)`: Returns a one-character string with the specified Unicode value.

### String methods

- `str.charAt(k)`: Returns a one-character string formed from the character at index position `k` in `str`.

- `str.charCodeAt(k)`: Returns the Unicode value for the character at index position `k` in `str`.

- `str.substring(p_1, p_2)`
- `str.substring(p_1)`: Returns a new string of characters beginning at `p_1` in `str` and extending up to but not including `p_2`. If `p_2` is missing, the new string continues through the end of the original string.

- `str.indexOf(pattern)`
- `str.indexOf(pattern, k)`: Searches the string `str` for the string `pattern`. The search starts at the beginning, or at index `k`, if specified. The function returns the first index at which `pattern` appears, or `-1` if it is not found.

- `str.lastIndexOf(pattern)`
- `str.lastIndexOf(pattern, k)`: Operates like `indexOf`, but searches backward from position `k`. If `k` is missing, `lastIndexOf` starts at the end of the string.

- `str.replace(old, new)`: Returns a copy of `str` with the first instance of `old` (if any) replaced by `new`.

- `str.split(pattern)`: Splits the string into an array of substrings by dividing it at instances of `pattern`. Arrays are discussed in Chapter 8.

- `str.toLowerCase()`: Returns a copy of `str` converting all characters to lowercase.

- `str.toUpperCase()`: Returns a copy of `str` converting all characters to uppercase.

### String methods available only in recent versions of JavaScript

- `str.startsWith(prefix)`: Returns `true` if `str` starts with the characters in `prefix`.

- `str.endsWith(suffix)`: Returns `true` if `str` ends with the characters in `suffix`.

- `str.trim()`: Returns a copy of `str` after removing whitespace from each end.
7.2 String operations

Determining the length of a string

The simplest operation that you can perform on a string value—which you already know from Chapter 2—is determining its length, which is the number of characters it contains. Given a JavaScript string variable `str`, you can determine the length by evaluating `str.length`. It is important to note that, in contrast to the other string operations listed in Figure 7-2, length is not defined as a method but instead as a JavaScript property, which is a data value associated with an object. Defining `length` as a property means that no parentheses appear after the property name.

As an example, if `ALPHABET` is defined as

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

the expression `ALPHABET.length` has the value 26. Similarly, if you create a variable `str` using the declaration

```javascript
let str = "";
```

the expression `str.length` has the value 0. The string containing no characters at all comes up frequently in programming and is called the empty string.

Selecting characters from a string

In JavaScript, positions within a string are numbered starting from 0. For example, the characters in `ALPHABET` are numbered as in the following diagram:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

The position number written underneath each character is called its index.

In JavaScript, you select a character from a string by calling the charAt method. For example, the expression

```javascript
ALPHABET.charAt(0)
```

selects the character "A" at the beginning, which is returned as a one-character string. Since character numbering in JavaScript begins at 0, the last character in a string appears at the index position that’s one less than the length of the string. Thus, you can select the "Z" at the end of `ALPHABET` with the following expression:

```javascript
ALPHABET.charAt(ALPHABET.length - 1)
```

JavaScript allows you to retrieve the underlying Unicode value of the character at any index position by calling charCodeAt with the desired index. For example,

```javascript
ALPHABET.charCodeAt(0)
```
returns the Unicode value of the character "A" at index position 0, which you can
determine from Figure 7-1 has the value $41_{10}$, or 65. To convert from character
codes to strings, you need to apply the function `String.fromCharCode`, which,
like the various `Math` functions, is associated with a class rather than an object.
Calling `String.fromCharCode(90)`, for example, returns the one-character string "Z".

**Extracting parts of a string**

While concatenation makes longer strings from shorter pieces, you often need to do
the reverse: separate a string into the shorter pieces it contains. A string that is part
of a longer string is called a *substring*. JavaScript’s string class exports a method
called `substring` that takes two parameters: the index of the first character you
want to select and the index of the character that immediately follows the desired
substring. For example, the method call

```
ALPHABET.substring(1, 4)
```

returns the three-character substring "BCD". Because indices in JavaScript begin at
0, the character at index position 1 is the character "B".

The `substring` method implements the following special cases:

- If the second argument is missing, `substring` extracts characters to the end of
  the string.
- If either argument is less than 0, it is assumed to be 0.
- If either argument is greater than the length of the string, it is assumed to be the
  length of the string.
- If the first argument is greater than the second, `substring` treats them as if they
  were supplied in the opposite order.

**Searching within a string**

From time to time, you will find it useful to search a string to see whether it
contains a particular character or substring. To support such search operations,
JavaScript’s string class exports a method called `indexOf`, which comes in two
forms. The simplest form of the call is

```
str.indexOf(pattern);
```

where *pattern* is the content you’re looking for. When called, the `indexOf` method
searches through `str` looking for the first occurrence of the pattern. If the search
value is found, `indexOf` returns the index position at which the match begins. If
the character does not appear before the end of the string, `indexOf` returns −1.
The `indexOf` method takes an optional second argument that indicates the index position at which to start the search. The effect of both styles of the `indexOf` method is illustrated by the following examples, which assume that the variable `str` contains the string "hello, world":

```
str.indexOf("o")    → 4
str.indexOf("o", 5) → 8
str.indexOf("o", 9) → -1
```

The JavaScript string class also includes a `lastIndexOf` method that works like `indexOf`, except that it searches backward for a match, starting from the specified index position or from the end of the string in the single-argument case.

The `indexOf` method makes it possible to determine whether a character fits into a specific class of characters in a particularly concise way. The following predicate function, for example, tests whether the character `ch` is one of the ten decimal digits:

```
function isDigit(ch) {
    return ch.length === 1 &&
    "0123456789".indexOf(ch) !== -1;
}
```

The first part of the Boolean expression in the `return` statement checks to see if `ch` is a single character, and the second part checks to see whether that character appears in the string "0123456789".

**Case conversion**

The methods `toLowerCase` and `toUpperCase` convert any alphabetic characters in the receiver string to the specified case, leaving any other characters unchanged. For example, if `str` contains "hello, world", calling `str.toUpperCase()` returns “HELLO, WORLD”. Similarly, calling `ALPHABET.toLowerCase()` returns "abcdefghijklmnopqrstuvwxyz".

It is important to remember that the methods in JavaScript’s `String` class do not change the value of the receiver but instead return an entirely new string value. Thus, calling `str.toUpperCase()` doesn’t change the value of the variable `str`. If you want to change the value of `str` to its uppercase equivalent, you need to use an assignment statement to store the value back into the variable, as in

```
str = str.toUpperCase();
```

The `toUpperCase` method makes it easy to write a predicate function called `equalsIgnoreCase` that checks whether two strings are equal if the comparison ignores the distinction between uppercase and lowercase characters, as follows:
function equalsIgnoreCase(s1, s2) {
    return s1.toUpperCase() === s2.toUpperCase();
}

The *startsWith*, *endsWith*, and *trim* methods

Modern versions of JavaScript include several methods that were not part of the original string library. The *startsWith* method returns `true` if the receiver string begins with the specified prefix. For example, the Boolean expression

```javascript
answer.startsWith("y") || answer.startsWith("Y")
```

is `true` if the string variable `answer` begins "y" or "Y". The *endsWith* method is symmetric and returns `true` if the string ends with the specified suffix. The *trim* method returns a copy of the string after removing any spaces from the beginning or end of the string.

Because these methods don’t exist in older versions of JavaScript, some web designers try to avoid them to ensure that their code runs on as many browsers as possible. You can, however, implement these tools as predicate functions. For example, the function

```javascript
function startsWith(str, prefix) {
    return prefix === str.substring(0, prefix.length);
}
```

### 7.3 Common string patterns

Even though the methods exported by JavaScript’s string class provide the tools you need to implement simple string applications, it is usually easier to write programs by adapting code patterns that implement particularly common operations. The two most important patterns are iterating through the characters in a string and growing a string by concatenation. These patterns are outlined in more detail in the sections that follow.

**Iterating through the characters in a string**

When you work with strings, one of the most important patterns involves iterating through the characters in a string, which requires the following code:

```javascript
for (let i = 0; i < str.length; i++) {
    . . . body of loop that uses the character str.charAt(i) . . .
}
```
On each loop cycle, the expression `str.charAt(i)` refers to the $i^{th}$ character in the string. Because the purpose of the loop is to process every character, the loop continues as long as $i$ is less than the length of the string. Thus, you can count the number of spaces in a string using the following function:

```javascript
function countSpaces(str) {
    let nSpaces = 0;
    for (let i = 0; i < str.length; i++) {
        if (str.charAt(i) === " ") nSpaces++;
    }
    return nSpaces;
}
```

For some applications, you will find it useful to iterate through a string in the opposite direction, starting with the last character and continuing backward until you reach the first. This style of iteration uses the following `for` loop:

```javascript
for (let i = str.length - 1; i >= 0; i--)
```

Here, the index $i$ begins at the last index position, which is one less than the length of the string, and then decreases by one on each cycle, down to and including the index position 0.

Assuming that you understand the syntax and semantics of the `for` statement, you could work out the patterns for each iteration direction from first principles each time this pattern comes up in an application. Doing so, however, would slow you down enormously. These iteration patterns are worth memorizing so that you don’t have to waste any time thinking about them. Whenever you recognize that you need to cycle through the characters in a string, some part of your nervous system between your brain and your fingers should be able to translate that idea effortlessly into the following line:

```javascript
for (let i = 0; i < str.length; i++)
```

## Growing a string through concatenation

The other string pattern that is important to memorize involves creating a new string one character at a time. The details of the loop depend on the application, but the general pattern for creating a string by concatenation looks like this:

```javascript
let str = " ";
for (whatever loop header line fits the application) {
    str += the next substring or character;
}
```
As a simple example, the following function returns a string consisting of \( n \) copies of the string \( \text{str} \):

\[
\text{function nCopies}(n, \text{str}) \{
\text{let result} = \"\";
\text{for (let i = 0; i < n; i++)} \{
\text{result += str;}
\}
\text{return result;}
\}
\]

The \text{nCopies} function is useful if, for example, you need to generate some kind of section separator in console output. One strategy to accomplish this goal would be to use the statement

\[
\text{console.log(nCopies(72, \"\-\"));}
\]

which prints a line of 72 hyphens.

\textbf{Combining the iteration and concatenation patterns}

Many string-processing functions use the iteration and concatenation patterns together. For example, the following function reverses the argument string so that, for example, calling \text{reverse("stressed")} returns \text{"desserts"}:

\[
\text{function reverse}(\text{str}) \{
\text{let result} = \"\";
\text{for (let i = str.length - 1; i >= 0; i--) \{
\text{result += str.charAt(i);} 
\}}
\text{return result;}
\}
\]

You could also implement \text{reverse} by running the loop in the forward direction and concatenating each new character to the front of the \text{result} string. If you use this strategy, the \text{for} loop looks like this:

\[
\text{for (let i = 0; i < str.length; i++) \{
\text{result = str.charAt(i) + result;}
\}}
\]

\textbf{7.4 String applications}

The easiest way to improve your understanding of strings is to look at several sample applications. The sections that follow walk you through four applications that use strings in different ways.
Checking for palindromes

A palindrome is a word that reads identically backward and forward, such as level or noon. The goal of this section is to write a predicate function `isPalindrome` that checks whether a string is a palindrome. Calling `isPalindrome("level")` should return true; calling `isPalindrome("xyz")` should return false.

As with most programming problems, there is more than one strategy for solving this problem. One strategy is to use a for loop to run through each index position in the first half of the string checking to see whether the character in that position matches the one in the symmetric position relative to the end of the string. Adopting that strategy leads to the following code:

```javascript
function isPalindrome(str) {
  let n = str.length;
  for (let i = 0; i < n / 2; i++) {
    if (str.charAt(i) != str.charAt(n - i - 1)) {
      return false;
    }
  }
  return true;
}
```

If you make use of the functions you already have, you can code `isPalindrome` in a much simpler form, as follows:

```javascript
function isPalindrome(str) {
  return str === reverse(str);
}
```

Of these implementations, the first is substantially more efficient. The second implementation constructs the reversed string by concatenation, which requires the creation of several new strings. The first version works by selecting and comparing characters, which are less costly operations.

Despite this difference in efficiency, the second version has many advantages, particularly as an example for new programmers. For one thing, it takes advantage of existing code by making use of the `reverse` function. For another, it hides the complexity involved in calculating index positions required by the first version. It takes at least a minute or two for most students to figure out why the code includes the selection expression `str.charAt(n - i - 1)` or why it is appropriate to use the < operator in the for loop test, as opposed to <=. By contrast, the line

```javascript
return str === reverse(str);
```
reads as fluidly as English: a string is a palindrome if it is equal to the same string if you reverse it. That, after all, is precisely the definition of a palindrome.

Particularly as you are learning about programming, it is much more important to work toward the clarity of the second implementation than the efficiency of the first. Given the speed of modern computers, it is almost always worth sacrificing some efficiency to make a program easier to understand.

**Generating acronyms**

An *acronym* is a new word formed by combining, in order, the initial letters of a series of words. For example, *NATO* is an acronym formed from the first letters in *North Atlantic Treaty Organization*. The goal of this section is to write a function called *acronym* that takes a string composed of separate words and returns its acronym. Thus, calling the method

```javascript
acronym("North Atlantic Treaty Organization")
```

should return the string "NATO".

When you first look at the problem, it might seem that the obvious approach is to start with the first character and then search for spaces in a *while* loop. Each time the function finds one, it can concatenate the next character onto the end of the string variable used to hold the result. When no more spaces appear in the string, the acronym is complete. This strategy can be translated into a JavaScript implementation as follows:

```javascript
function acronym(str) {
    let result = str.charAt(0);
    let sp = str.indexOf(" ");
    while (sp !== -1) {
        result += str.charAt(sp + 1);
        sp = str.indexOf(" ", sp + 1);
    }
    return result;
}
```

Although this implementation works for some strings, it fails for others. For example, it produces the correct algorithm only if each pair of words is separated by exactly one space. If some of the words are separated using hyphens—as in "*self-contained underwater breathing apparatus*" for which the acronym is "*scuba*"—this implementation will fail to return the correct result. Worse still, the function will generate an error condition if the word ends with a space, because the code will try to select the following character, which doesn’t exist.
The following implementation may at first seem harder to follow, but it correctly handles the special cases in which the earlier version fails:

```javascript
function acronym(str) {
  let result = "";
  let inWord = false;
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);
    if (isLetter(ch)) {
      if (!inWord) result += ch;
      inWord = true;
    }
    else {
      inWord = false;
    }
  }
  return result;
}
```

This implementation uses the standard idiom to go through the string character by character looking at each character. It determines the word boundaries by maintaining a Boolean variable `inWord`, which is `true` if the process is scanning letters and `false` if it is scanning nonletters. New letters get added to the acronym only when the code sees a letter when it was previously not scanning a word.

### Translating English to Pig Latin

To give you more of a sense of how to implement string-processing applications, this section describes a JavaScript function that takes a line of text and translates each word in that line from English to Pig Latin, a made-up language familiar to most children in the English-speaking world. In Pig Latin, words are formed from their English counterparts by applying the following rules:

1. If the word contains no vowels, no translation is done, which means that the Pig Latin word is the same as the original.
2. If the word begins with a vowel, the Pig Latin translation consists of the original word followed by the suffix `way`.
3. If the word begins with a consonant, the Pig Latin translation is formed by extracting the string of consonants up to the first vowel, moving that collection of consonants to the end of the word, and then adding the suffix `ay`.

As an example, suppose that the English word is `scram`. Because the word begins with a consonant, you divide it into two parts: one consisting of the letters before the first vowel and one consisting of that vowel and the remaining letters:
You then interchange these two parts and add \textit{ay} at the end, as follows:

Thus the Pig Latin word for \textit{scram} is \textit{amsray}. For a word that begins with a vowel, such as \textit{apple}, you simply add \textit{way} to the end, which leaves you with \textit{appleway}.

The code for \texttt{PigLatin.js} appears in Figure 7-3. The file exports two functions for clients to use. The \texttt{wordToPigLatin} function converts a word to its Pig Latin equivalent. The \texttt{toPigLatin} function takes a line of text and converts the entire line to Pig Latin by divides the line into words and then converting each word. Characters that are not part of a word are copied directly to the output line so that punctuation and spacing remain unaffected.

![Functions to translate English to Pig Latin](image.png)
/*
 * Translates a word to Pig Latin using the following rules:
 * 1. If the word begins with a vowel, add "way" to the end of the word.
 * 2. If the word begins with a consonant, extract the leading consonants
 *      up to the first vowel, move them to the end, and then add "ay".
 * 3. If the word contains no vowels, return the word unchanged.
 */

function wordToPigLatin(word) {
    let vp = findFirstVowel(word);
    if (vp === -1) {
        return word;
    } else if (vp === 0) {
        return word + "way";
    } else {
        let head = word.substring(0, vp);
        let tail = word.substring(vp);
        return tail + head + "ay";
    }
}

/*
 * Returns the index of the first vowel in the word, or -1 if none.
 */

function findFirstVowel(word) {
    for (let i = 0; i < word.length; i++) {
        if (isEnglishVowel(word.charAt(i))) return i;
    }
    return -1;
}

/*
 * Returns true if the character ch is a letter.
 */

function isLetter(ch) {
    return ch.length === 1 && ((ch >= "A" && ch <= "Z") ||
                               (ch >= "a" && ch <= "z"));
}

/*
 * Returns true if the character ch is a vowel (A, E, I, O, or U, in
 * either upper or lower case).
 */

function isEnglishVowel(ch) {
    return ch.length === 1 && "AEIOUaeiou".indexOf(ch) !== -1;
}
The following console log gives a few examples of the functions `toPigLatin` and `wordToPigLatin`:

```
JavaScript Console
> toPigLatin("this is pig latin.")
isthay isway igpay atinlay.
> wordToPigLatin("scram")
amscry
> wordToPigLatin("apple")
appleway
> 
```

It is worth taking a careful look at the implementations of `toPigLatin` and `wordToPigLatin` in Figure 7-3. The `toPigLatin` function finds the word boundaries in the input, which provides a useful pattern for separating a string into individual words. The `wordToPigLatin` function uses `substring` to extract pieces of the English word and then uses concatenation to put them back together in their Pig Latin form.

### Implementing simple ciphers

Codes and ciphers have been around in some form or another for most of recorded history. There is evidence to suggest that coded messages were used in ancient Egypt, China, and India, possibly as early as the third millennium BCE, although few details of the cryptographic systems have survived. In Book 6 of the *Iliad*, Homer suggests the existence of a coded message when King Proitos, seeking to have the young Bellerophontes killed, has

> . . . sent him to Lykia, and handed him murderous symbols, which he inscribed on a folding tablet, enough to destroy life

Shakespeare’s *Hamlet*, of course, has Rosencrantz and Guildenstern carry a similarly dangerous missive, but Hamlet’s message is secured under a royal seal. In the *Iliad*, there is nothing to suggest that Bellerophon cannot see the “murderous symbols,” which implies that their meaning must somehow be disguised.

One of the first encryption systems whose details survive is the **Polybius square**, developed by the Greek historian Polybius in the second century BCE. In this system, the letters of the alphabet are arranged to form a 5×5 grid in which each letter is represented by its row and column number. Suppose, for example, that you want to transmit following English version of Pheidippides’ message to Sparta:

THE ATHENIANS BESEECH YOU TO HASTEN TO THEIR AID

This message can be transmitted as a series of numeric pairs, as follows:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
</tr>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td></td>
</tr>
</tbody>
</table>

```
44 23 15 11 44 23 15 33 24 11 33 43 12 15 43 15 15 13 23 54
34 45 44 34 23 11 43 44 15 33 44 44 23 15 24 42 11 24 14
```
The advantage of the Polybius square is not so much that it allows for secret messages, but that it simplifies the problem of transmission. Each letter in the message can be represented by holding between one and five torches in each hand, which allows a message to be communicated visually over a great distance. By reducing the alphabet to an easily transmittable code, the Polybius square anticipates such later developments as Morse code and semaphore, not to mention modern digital encodings such as ASCII or Unicode.

In *De Vita Caesarum*, written sometime around 110 CE, the Roman historian Suetonius describes an encryption system used by Julius Caesar, as follows:

If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.

Even today, the technique of encoding a message by shifting letters a certain distance in the alphabet is called a *Caesar cipher*. According to the passage from Suetonius, each letter is shifted three positions ahead in the alphabet. For example, if Caesar had had time to translate his final words according to his coding system, *ET TU BRUTE* would have come out as *HW WX EUXWH*, because *E* gets moved three letters ahead to *H*, *T* gets moved three to *W*, and so on. Letters that get advanced past the end of the alphabet wrap around back to the beginning, so that *X* would become *A*, *Y* would become *B*, and *Z* would become *C*.

The *caesarCipher* function in Figure 7-4 translates the letters in a string according to the rules for constructing a Caesar cipher. The code uses `charCodeAt` to convert characters into their Unicode values and then uses the remainder operator to implement the cyclical shift that wraps around to the beginning of the alphabet. Once it has computed the new character code, it uses `String.fromCharCode` to convert the Unicode value back into a string.

The following console log demonstrates the operation of *caesarCipher*:

```
> caesarCipher("This is a secret message.", 13)
Guvf vf n frperg zrffntr.
> caesarCipher("Guvf vf n frperg zrffntr.", -13)
This is a secret message.
> caesarCipher("IBM 9000", -1)
HAL 9000
> 
```
The graphical programs you have seen in this chapter use interactivity in the way that modern computers typically do. The programs create graphical displays and then let a user interact with those displays by moving a mouse. That style of interaction, however, is relatively new in the history of computing. Prior to the development of GUI technology at Xerox PARC and its later commercialization in the Apple Macintosh, most interaction with the computer was accomplished through the keyboard. In 1999, the science-fiction writer Neal Stephenson wrote an essay entitled “In the beginning was the command line,” in which he fondly recalls his memories of those earlier days.

Even though it may seem like ancient history, there are still situations in which reading data or commands from the console offers a useful style of human-computer interaction. If nothing else, it is often easier to test your functions by writing simple console-based based programs that let you see the results of making particular calls. In fact, that’s exactly what you’ve been doing when you use the JavaScript console. You type in expressions and get to see the results.
Console-based programs are also useful for illustrating programming principles. In my previous books designed to introduce the art and science of programming, one of the examples I used was a program that read a list of integers from the user and printed out their sum, as shown in the following console script:

```
> AddIntegerList();
Enter a list of integers up to a blank line.
? 1
? 2
? 3
? 4
? The sum is 10.
>
```

The `AddIntegerList` function begins by giving instructions to the user and then asks the user to enter integers on the console, one per line. When the user enters a blank line after the last question mark, the program displays the sum.

Programs of this sort are surprisingly difficult to write in JavaScript. JavaScript depends on an event-driven model, even for interaction with the console. It is therefore impossible for a program to print a question mark, wait for the user to enter an integer, and then continue on with the program. A JavaScript program must instead declare its interest in being notified when a line appears by supplying a callback function. That callback function is then responsible for processing the input and taking whatever actions are necessary to produce the desired result.

The code for `AddIntegerList.js` appears in Figure 7-14. The first new feature used in the program is the `requestInput` method in the `console` object. The first argument is a prompt, which is used to notify the user that some input is expected. The second argument is the callback function to invoke when that input appears. The callback function takes one argument, which is the input line. If that string is empty, `processLine` displays the sum of the values entered; otherwise, `processLine` converts the string to an integer and adds it to the running total.

JavaScript includes two built-in functions, `parseInt` (which you’ve already seen earlier in this chapter) and `parseFloat`, which convert strings to numbers. The first requires that the string be an integer; the second (the name comes from the term floating-point used to refer to numbers with fractional parts) reads any number. These methods signal failure by returning the constant `NaN`, which is short for “not a number.” Because there is no reason to suppose that two computations that produce `NaN` are producing the same value, JavaScript defines `NaN` so that it is not equal to itself; the only way to check for `NaN` is to call the built-in function `isNaN`. The code for `processLine` displays an error message if the user fails to enter a valid integer. If the number is valid, `processLine` adds that integer to `sum`. 

In this chapter, you have learned how to use the `String` class, which makes it possible to write string-processing functions without worrying about the details of the underlying representation. The important points in this chapter include:

- The fundamental unit of information in a modern computer is a *bit*, which can be in one of two possible states. The state of a bit is usually represented in memory diagrams using the binary digits 0 and 1, but it is equally appropriate to think of these values as *off* and *on* or *false* and *true*, depending on the application.

- Sequences of bits are combined inside the hardware to form larger structures, including *bytes*, which are eight bits long, and *words*, which are large enough to contain an integer.

- Computer scientists tend to record the values of bit sequences in *hexadecimal* (base 16), which allows binary values to be represented in a more compact form.

- Numbers don’t have bases; representations do.

- Nonnumeric data values are represented by numbering the elements in the domain and then using those numbers as codes for the original values.
• Characters are represented internally using a coding scheme called *Unicode*, which assigns numeric values to characters from a wide range of languages.

• The *String* class represents a type that is conceptually a sequence of characters. The characters positions in a string are assigned index numbers that start at 0 and extend up to one less than the length of the string.

• The most common methods exported by the *String* class appear in Figure 7-2 on page 228. Because *String* is a class, the methods use the receiver syntax instead of a more traditional functional form.

• The standard pattern for iterating through the characters in a string is
  
  ```javascript
  for (let i = 0; i < str.length; i++) {
    . . . body of loop that manipulates str.charAt(i) . . .
  }
  ```

• The standard pattern for growing a string by concatenation is
  
  ```javascript
  let str = "";
  for (whatever loop header line fits the application) {
    str += the next substring or character;
  }
  ```

### Review questions

1. Define the following terms: *bit*, *byte*, and *word*.

2. What is the etymology of the word *bit*?

3. Convert each of the following decimal numbers to its hexadecimal equivalent:
   
   - a. 17
   - b. 256
   - c. 1729
   - d. 2766

4. Convert each of the following hexadecimal numbers to decimal:
   
   - a. 17
   - b. 64
   - c. CC
   - d. FAD

5. What JavaScript functions allow you to convert between numbers and their corresponding string representations?

6. In your own words, state the principle of enumeration.

7. What does *ASCII* stand for?

8. What is the relationship between *ASCII* and *Unicode*?
9. By consulting Figure 7-1, determine the Unicode values of the characters "$", 
"@", "0", and "x".

10. True or false: In JavaScript, you can determine the length of the string stored in
the variable `str` by calling `length(str)`.

11. True or false: The index positions in a string begin at 0 and extend up to the
length of the string minus 1.

12. Which methods in Figure 7-2 are defined only in recent versions of JavaScript?

13. How do you extract the character at position `k` in a string? How would you
determine the Unicode value of that character?

14. What are the arguments to the `substring` method? What happens if you omit
the second argument?

15. What is *lexicographic ordering*?

16. What value does `indexOf` return if the pattern string does not appear?

17. What is the significance of the optional second argument to `indexOf`?

18. Suppose that you have declared and initialized the variable `s` as follows:
```javascript
let s = "hello, world";
```
Given that declaration, what is the effect of each of the following calls:

a. `s + "!"`

b. `s.length`

c. `s.charAt(5)`

d. `s.indexOf("l")`

e. `s.indexOf("l", 5)`

f. `s.replace("h", "j")`

g. `s.substring(0, 3)`

h. `s.substring(7)`
i. `s.substring(3, 5)`

j. `s.substring(3, 3)`

19. What is the pattern for iterating through each character in a string?

20. How does the pattern in question 19 change if you want to iterate through the
characters backwards, starting with the last character and ending with the first?

21. What is the pattern for growing a string through concatenation?

---

### Exercises

1. In exercise 18 from Chapter 4, you wrote a program to find perfect numbers.
Rewrite that program so that it also displays the binary form of these numbers.
As you can see if you run this program, the first few perfect numbers follow an
interesting pattern if you write them out in binary. Euclid discovered this pattern more than 2000 years ago, and the 18th-century Swiss mathematician Leonhard Euler proved that all even perfect numbers display this pattern.

2. As noted in the discussion of the built-in method in the String class, the `startsWith` and `endsWith` methods are not implemented in many browsers because they were standardized only recently. Using the `startsWith` function on page 203 as a model, implement the function `endsWith(str, suffix)` that checks whether `str` ends with the specified suffix.

3. Implement the function `isEnglishConsonant(ch)`, which returns `true` if `ch` is a consonant in English, that is, any alphabetic character except one of the five vowels: "a", "e", "i", "o", and "u". As with the `isEnglishVowel` function presented in the text, your method should recognize both lower- and uppercase consonants.

4. Write a function `randomWord` that returns a randomly constructed “word” consisting of randomly chosen lowercase letters. The number of letters in the word should also be chosen randomly by picking a number between the values of the constants `MIN_LETTERS` and `MAX_LETTERS`.

5. Implement a function `capitalize(str)` that returns a string in which the initial character is capitalized (if it is a letter) and all other letters are converted to lower case. Characters other than letters are not affected. For example, both `capitalize("BOOLEAN")` and `capitalize("boolean")` should return the string "Boolean".

6. Write a method `createDateString(day, month, year)` that returns a string consisting of the day of the month, a hyphen, the first three letters in the name of the month, another hyphen, and the last two digits of the year. For example, calling `createDateString(22, 11, 1963)` should return the string "22-Nov-63".

7. In many word games, letters are scored according to their point values, which are inversely proportional to their frequency in English words. In Scrabble™, the points are allocated as follows:

<table>
<thead>
<tr>
<th>Points</th>
<th>Letters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A, E, I, L, N, O, R, S, T, U</td>
</tr>
<tr>
<td>2</td>
<td>D, G</td>
</tr>
<tr>
<td>3</td>
<td>B, C, M, P</td>
</tr>
<tr>
<td>4</td>
<td>F, H, V, W, Y</td>
</tr>
<tr>
<td>5</td>
<td>K</td>
</tr>
<tr>
<td>8</td>
<td>J, X</td>
</tr>
<tr>
<td>10</td>
<td>Q, Z</td>
</tr>
</tbody>
</table>
For example, the word “FARM” is worth 9 points in Scrabble: 4 for the F, 1 each for the A and the R, and 3 for the M. Write a function `scrabbleScore` that takes a word and returns its score in Scrabble, not counting any of the other bonuses that occur in the game. You should ignore any characters other than uppercase letters in computing the score.

8. The concept of a palindrome is often extended to full sentences by ignoring punctuation and differences in the case of letters. For example, the sentence

   Madam, I’m Adam.

is a sentence palindrome, because if you look only at the letters and ignore any case distinctions, it reads identically backward and forward.

Write a predicate function `isSentencePalindrome(str)` that returns `true` if `str` fits this definition of a sentence palindrome. For example, you should be able to use your function to reproduce the following console session:

   JavaScript Console
   > isSentencePalindrome("Madam, I'm Adam.")
   true
   > isSentencePalindrome("Able was I ere I saw Elba.")
   true
   > isSentencePalindrome("Not a palindrome.")
   false
   >

9. Write a function `createRegularPlural(word)` that returns the plural of `word` formed by following these standard English rules:
   a. If the word ends in s, x, z, ch, or sh, add es to the word.
   b. If the word ends in a y preceded by a consonant, change the y to ies.
   c. In all other cases, add just an s.

Design a set of test cases to verify that your function works.

10. In English, the notion of an ongoing action is expressed using the present progressive tense, which involves the addition of an ing suffix to the verb. For example, the sentence I think conveys a sense that one is capable of thinking; by contrast, the sentence I am thinking conveys the impression that one is currently doing so. The ing form of the verb is called the **present participle**.

    Unfortunately, creating the present participle is not always as simple as adding the ing ending. One common exception is words that end in a silent e, such as cogitate. In such cases, the e is usually dropped, so that the participle form becomes cogitating. Another common exception involves words that end with a single consonant, which typically gets doubled in the participle form. For example, the verb run becomes running.
Although there are many exceptions, you can construct a large fraction of the legal participle forms in English by applying the following rules:

a) If the word ends in an e preceded by a consonant, take the e away before adding ing. Thus, move should become moving. If the e is not preceded by a consonant, it should remain in place, so that see becomes seeing.

b) If the word ends in a consonant preceded by a vowel, insert an extra copy of that consonant before adding ing. Thus, jam should become jamming. If, however, there is more than one consonant at the end of the word, no such doubling takes place, so that walk becomes walking.

c) In all other circumstances, simply add the ing suffix.

Write a function `createPresentParticiple(verb)` that takes an English verb, which you may assume is entirely lowercase and at least two characters long, and forms the participle using these rules.

11. As in most languages, English includes two types of numbers. The **cardinal numbers** (such as one, two, three, and four) are used in counting; the **ordinal numbers** (such as first, second, third, and fourth) are used to indicate a position in a sequence. In text, ordinals are usually indicated by writing the digits in the number, followed by the last two letters of the English word that names the corresponding ordinal. Thus, the ordinal numbers first, second, third, and fourth often appear in print as 1st, 2nd, 3rd, and 4th. The ordinals for 11, 12, and 13, however, are 11th, 12th, and 13th. Devise a rule that determines what suffix should be added to each number, and then use this rule to write a function `createOrdinalForm(n)` that returns the ordinal form of the number `n` as a string.

12. The waste of time in spelling imaginary sounds and their history (or etymology as it is called) is monstrous in English . . .

—George Bernard Shaw, 1941

In the early part of the 20th century, there was considerable interest in both England and the United States in simplifying the rules used for spelling English words, which has always been a difficult proposition. One suggestion advanced as part of this movement was to eliminate all doubled letters, so that bookkeeper would be written as bokeper and committee would become comite. Write a function `removeDoubledLetters(str)` that returns a new string in which any duplicated characters in `str` have been replaced by a single copy.

13. When large numbers are written on paper, it is traditional—at least in the United States—to use commas to separate the digits into groups of three. For
example, the number one million is usually written as 1,000,000. Implement a
function

```javascript
function addCommas(digits)
```

that takes a string of decimal digits representing a number and returns the string
formed by inserting commas at every third position, starting on the right. Your
implementation of the `addCommas` function should be able to reproduce the
following console log:

<table>
<thead>
<tr>
<th>JavaScript Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; addCommas(&quot;17&quot;)</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>&gt; addCommas(&quot;2001&quot;)</td>
</tr>
<tr>
<td>2,001</td>
</tr>
<tr>
<td>&gt; addCommas(&quot;12345678&quot;)</td>
</tr>
<tr>
<td>12,345,678</td>
</tr>
<tr>
<td>&gt; addCommas(&quot;999999999&quot;)</td>
</tr>
<tr>
<td>999,999,999</td>
</tr>
</tbody>
</table>

14. As written, the `PigLatin` program in Figure 7-3 behaves oddly if you enter a
string that includes words beginning with an uppercase letter. For example, if
you were to capitalize the first word in the sentence and the name of the Pig
Latin language, you would see the following output:

<table>
<thead>
<tr>
<th>JavaScript Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; toPigLatin(&quot;This is Pig Latin.&quot;)</td>
</tr>
<tr>
<td>IsThey isway igPay atinLay.</td>
</tr>
</tbody>
</table>

Rewrite the `wordToPigLatin` function so that any word that begins with a
capital letter in the English line still begins with a capital letter in Pig Latin.
Thus, after you make the necessary changes in the program, the output should
look like this:

<table>
<thead>
<tr>
<th>JavaScript Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; toPigLatin(&quot;This is Pig Latin.&quot;)</td>
</tr>
<tr>
<td>Isthay isway IgPay Atinlay.</td>
</tr>
</tbody>
</table>

15. Most people—at least those in English-speaking countries—have played the
Pig Latin game at some point in their lives. There are other invented
“languages” in which words are created using some simple transformation of
English. One such language is called `Obenglobish`, in which words are created
by adding the letters `ob` before the vowels (`a`, `e`, `i`, `o`, and `u`) in an English word.
For example, under this rule, the word `english` gets the letters `ob` added before
the `e` and the `i` to form `obenglish`, which is how the language got its name.
In official Obenglobish, the ob characters are added only before vowels that are pronounced, which means that a word like game would become gobame rather than gobamobe because the final e is silent. While it is impossible to implement this rule perfectly, you can do a pretty good job by adopting the rule that the ob should be added before every vowel in the English word except

- Vowels that follow other vowels
- An e that occurs at the end of the word

Write a function obenglobish that takes an English word and returns its Obenglobish equivalent, using the translation rule given above. Your function should allow you to generate the following sample run:

```
> toObenglobish("english")
obenglish
> toObenglobish("hobnob")
hobbobobob
> toObenglobish("gooliest")
gobooliest
> toObenglobish("amaze")
obamobase
> toObenglobish("rot")
robot
```

16. **There is no gene for the human spirit.**

—Tagline for the 1997 film GATTACA

The genetic code for all living organisms is carried in its DNA—a molecule with the remarkable capacity to replicate its own structure. The DNA molecule itself consists of a long strand of chemical bases wound together with a similar strand in a double helix. DNA’s ability to replicate comes from the fact that its four constituent bases—adenosine, cytosine, guanine, and thymine—combine with each other only in the following ways:

- Cytosine on one strand links only with guanine on the other, and vice versa.
- Adenosine links only with thymine, and vice versa.

Biologists abbreviate the names of the bases by writing only the initial letter: A, C, G, or T.

Inside the cell, a DNA strand acts as a template to which other DNA strands can attach themselves. As an example, suppose that you have the following DNA strand, in which the position of each base has been numbered as it would be in a JavaScript string:
Your mission in this exercise is to determine where a shorter DNA strand can attach itself to the longer one. If, for example, you were trying to find a match for the strand

the rules for DNA dictate that this strand can bind to the longer one only at position 1:

By contrast, the strand

matches at either position 2 or position 7.

Write a function

```
function findDNAMatch(s1, s2, start)
```

that returns the first position at which the DNA strand `s1` can attach to the strand `s2`. As in the `indexOf` method, the optional `start` parameter indicates the index position at which the search should start. If there is no match, `findDNAMatch` should return `-1`.

17. Although Caesar ciphers are simple, they are also extremely easy to break. A somewhat more secure scheme is to allow each letter in the message to be represented consistently by some other letter, but not one chosen by shifting the character a fixed distance in the alphabet. This kind of coding scheme is called a **letter-substitution cipher**.

The key in a letter-substitution cipher is a 26-character string that shows the enciphered counterpart of each of the 26 letters of the alphabet. For example, if the communicating parties choose "QWERTYUIOPASDFGHJKLZXCVBNM" as the key (which is unimaginatively generated by typing the letter keys on the keyboard in order), that key then corresponds to the following mapping:
Write a function `encrypt` that takes a string and a 26-character key and returns the string after applying a letter-substitution cipher with that key. For example, your function should be able to produce the following sample run:

```
JavaScript Console
> const KEY = "QWERTYUIOPASDFGHJKLMZXCVBNM";
> encrypt("Squeamish Ossifrage", KEY)
Ljxtqdoli Gilloykqut
```

The words *squeamish ossifrage* were part of the solution to a cryptographic puzzle designed by the inventors of the RSA encryption algorithm and then published in *Scientific American*.

18. Write a predicate function `isKeyLegal`, which takes as string and returns `true` if that string would be a legal key in a letter-substitution cipher. A key is legal only if it meets the following two conditions:

1. The key is exactly 26 characters long.
2. Every uppercase letter appears in the key.

These conditions automatically rule out the possibility that the key contains invalid characters or duplicated letters. After all, if all 26 uppercase letters appear and the string is exactly 26 characters long, there isn’t room for anything else.

19. Letter-substitution ciphers require the sender and receiver to use different keys: one to encrypt the message and one to decrypt it when it reaches its destination. Your task in this exercise is to write a function `invertKey` that takes an encryption key and returns the corresponding decryption key. In cryptography, that operation is called *inverting* the encryption key.

The idea of inverting a key is most easily illustrated by example. Suppose, for example, that the key is "QWERTYUIOPASDFGHJKLMZXCVBNM" as in the preceding problems Part 1. That key represents the following translation rule:

```
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓→
corresponding letter in the original must have been K. Similarly, the only way to get a B in the encrypted message is to start with an X in the original one. The first two entries in the inverted translation table therefore look like this:

\[
\begin{array}{cc}
A & B \\
\downarrow & \downarrow \\
K & X
\end{array}
\]

If you continue this process by finding each letter of the alphabet on the bottom of the original translation table and then looking to see what letter appears on top, you will eventually complete the inverted table, as follows:

\[
\text{ABCDEFGHIJKLMNOPQRSTUVWXYZ}
\]

\[
\begin{array}{cc}
A & B \\
\downarrow & \downarrow \\
K & X
\end{array}
\]

The inverted key is simply the 26-character string on the bottom row, which in this case is "KXVMCNOPHQRSZYIJADLEGWBUFT".

12. Write a program that reads in a list of integers on the console until the user enters a blank line. When the sentinel appears, your program should display the largest value in the list.

13. For a slightly more interesting challenge, write a program that finds both the largest and the second-largest integer in a list, prior to the entry of the blank line that ends the input. A sample run of this program might look like this:

```
> FindTwoLargest();
Enter a list of integers up to a blank line.
? 223
? 251
? 317
? 636
? 766
? 607
? 607
The largest value is 766.
The second-largest value is 636.
>
```

The input values in this example are the number of pages in the British hardcover editions of J. K. Rowling’s Harry Potter series. The output therefore tells us that the longest book (Harry Potter and the Order of the Phoenix) has 766 pages and the second-longest book (Harry Potter and the Goblet of Fire) weighs in at a mere 636 pages.
CHAPTER 8

Arrays

I’m not rich because I invented VisiCalc, but I feel that I’ve made a change in the world. That’s a satisfaction money can’t buy.

— Dan Bricklin, November 1985, as quoted in Robert Slater, Portraits in Silicon

Bob Frankston and Dan Bricklin

In modern computing, one of the most visible applications of the array structure described in this chapter is the electronic spreadsheet, which uses a two-dimensional array to store tabular data. The first electronic spreadsheet was VisiCalc, which was released in 1979 by Software Arts, Incorporated, a small startup company founded by MIT graduates Dan Bricklin and Bob Frankston. VisiCalc proved to be a popular application, leading many larger firms to develop competing products, including Lotus 1 2 3 and, more recently, Microsoft Excel.
Up to now, the programs in this book have worked with individual data items. The real power of computing, however, comes from the ability to work with collections of data. This chapter introduces the idea of an array, which is an ordered collection of values. Arrays are important in programming largely because such collections occur quite often in the real world. Whenever you want to represent a set of values in which it makes sense to think about those values as forming a sequence, arrays are likely to play a role in the solution.

### 8.1 Introduction to arrays

An array is a collection of individual values in which the elements are identified by a sequential position. You must be able to list the individual values of an array in order: here is the first, here is the second, and so on. Conceptually, it is easiest to think of an array as a sequence of boxes, with one box for each data value in the array. Each of the values in an array is called an element.

Like every other data type in JavaScript, arrays can be stored in variables, passed as arguments to a function, and returned from functions as a result. And like every other data type, arrays in JavaScript support a set of operations appropriate to the type. For arrays, that set of operations allows you to manipulate both the contents and the ordering of elements. These operations are outlined in the sections that follow.

#### JavaScript array notation

Creating an array in JavaScript is much easier than it is in most other programming languages. All you need to do is list the elements of the array surrounded by square brackets and separated by commas. For example, the following declaration contains the numbers that correspond to coins in the United States:

```javascript
const COINS = [ 1, 5, 10, 25, 50, 100 ];
```

After making this definition, the value of the constant `COINS` is an array that corresponds to the following box diagram:

```
  COINS
      1   5   10  25  50  100
      0   1   2   3   4   5
```

The small numbers underneath the boxes in this diagram represent the position of that value in the array, which is called its index. When you use JavaScript’s array notation, the index numbers always begin with 0 and run up to one less than the number of elements. Thus, in an array with six elements, the index numbers are 0, 1, 2, 3, 4, and 5, as the preceding diagram shows.
Every JavaScript array has a field called `length` that contains the number of elements. The expression

```javascript
COINS.length
```

therefore has the value 6.

The elements of an array need not be numbers but can instead be any JavaScript value. For example, the following variable declaration defines `hogwarts` as an array containing the names of the four houses at the Hogwarts School of Witchcraft and Wizardry from J. K. Rowling’s Harry Potter novels:

```javascript
let hogwarts = [
    "Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"
];
```

The box diagram for this array looks like this:

```
    hogwarts
    "Gryffindor" "Hufflepuff" "Ravenclaw" "Slytherin"
```

The expression `hogwarts.length` has the value 4.

**Array selection**

To refer to a specific element within an array, you specify both the array name and the index corresponding to the position of that element within the array. The process of identifying a particular element within an array is called selection, and is indicated in JavaScript by writing the name of the array and following it with the index written in square brackets. For example, given the array definitions from the preceding section, the expression `COINS[2]` is 10, because that is the value at index 2 in the `COINS` array. Similarly, `hogwarts[0]` has the value "Gryffindor". If you select an index position that falls outside the limits of an array, JavaScript returns the value `undefined`.

The result of a selection expression is assignable, in the sense that you can use a selection expression on the left side of an assignment statement. For example, if some future Hogwarts leaders decided that they might need to honor a more worthy wizard, evaluating the expression

```javascript
hogwarts[3] = "Dumbledore";
```

would change the value of the `hogwarts` array to

```
    hogwarts
    "Gryffindor" "Hufflepuff" "Ravenclaw" "Dumbledore"
```
Arrays are often used in connection with for loops that step through every index position in the array. The usual pattern for doing so is analogous to the pattern for iterating through the characters in a string presented in Chapter 7:

```javascript
for (let i = 0; i < array.length; i++) {
  . . . body of loop that uses the element array[i] . . .
}
```

A simple example of the use of for loops with arrays is the function

```javascript
function listArray(array) {
  for (let i = 0; i < array.length; i++) {
    console.log(array[i]);
  }
}
```

This function simply lists the elements of array, one per line, on the console. For example, after defining this function, you could generate the following console session:

```
> let hogwarts = [
  "Gryffindor",
  "Hufflepuff",
  "Ravenclaw",
  "Slytherin"
];
> hogwarts
["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]
> listArray(hogwarts);
Gryffindor
Hufflepuff
Ravenclaw
Slytherin
```

As a second example, the function

```javascript
function sumArray(array) {
  let sum = 0;
  for (let i = 0; i < array.length; i++) {
    sum += array[i];
  }
  return sum;
}
```

returns the sum of the elements in the array. The following console log shows several examples of sumArray in action:
The `reverseArray` function in Figure 8-1 offers yet another example of a function that takes an array parameter. The effect of this function is to reverse the order of the elements in the array passed in by the caller, as illustrated by the following console log:

```javascript
const COINS = [1, 5, 10, 25, 50, 100];
sumArray(COINS)
191
sumArray([])
0
sumArray([1, 2, 3, 4])
10
```

**Figure 8-1**  Function to reverse an array in place

```javascript
/*
 * File: ReverseArray.js
 * ----------------------
 * This file exports the function reverseArray, which reverses the
 * elements of an array.
 */

/*
 * Reverses the elements in the array. The change is reflected in the
 * argument array because it is passed as a reference and therefore
 * shares the same elements.
 */

function reverseArray(array) {
    for (let lh = 0; lh < Math.floor(array.length / 2); lh++) {
        let rh = array.length - 1 - lh;
        let tmp = array[lh];
        array[lh] = array[rh];
        array[rh] = tmp;
    }
}
```
It is worth spending a few minutes going through the code for `reverseArray`. The overall strategy is to use a `for` loop to go through the index positions in the first half of the array, marking that position with the variable `lh`, where the variable name is chosen to suggest the idea of a position on the left side of the array that you might point to with your left hand. The first statement in the loop body calculates the index position of the corresponding element at the right side of the array. The last three lines of the loop body interchange these two array elements using a temporary variable to ensure that no values are completely lost after the first assignment.

**Passing arrays as references**

The `reverseArray` function in Figure 8-1 raises a puzzling question if you think carefully about the code in light of the rules for parameter passing presented in Chapter 5. In the list of rules presented in the section entitled “The steps in calling a function,” rule 3 begins like this:

3. The value of each argument is copied into the corresponding parameter variable.

It is interesting to ask whether this rule is applied when the console script calls `reverseArray`. Is the `hogwarts` array copied into the parameter variable `array`? If so, why doesn’t the `reverseArray` function reverse the elements of the copy, leaving the original value of `hogwarts` unchanged?

The key to answering this question lies in understanding that the value of a JavaScript array is not the sequence of elements itself but is instead a reference to those elements, which indicates where those elements are stored in the computer’s memory. When you pass an array as an argument to a function, JavaScript copies the reference but does not copy the actual element values. The effect of this strategy is that a function and its caller have access to the same elements.

The idea that arrays—and indeed all JavaScript objects as you will learn in Chapter 9—are passed as references is so important that it is worth going through an example in more detail. Suppose that you have defined the following function:

```javascript
function testReverseArray() {
    let numbers = [1, 2, 3, 4, 5];
    reverseArray(numbers);
    console.log(numbers);
}
```

Calling this function creates a new stack frame and declares the variable `numbers`. After initializing the variable to an array value, the stack frame looks like this:
The important thing to note is that the elements of the array are stored outside the frame. The reference stored in `numbers` indicates where the array is stored. When the program calls `reverseArray`, the new stack frame looks like this:

The elements of the array are still in the same place in the computer’s memory as they were in the frame for `testReverseArray`. When `reverseArray` returns after completing its operation, the values in the `numbers` array will be reversed, as follows:

**Array operations**

In addition to selection, JavaScript offers a range of additional operations on arrays that are implemented as methods. The most common array methods are listed in Figure 8-2. This chapter provides complete details for less than half of these
Arrays

but you may still find them useful in writing your own code. One of the most important skills you need to master on your way to becoming a programmer is knowing how to look up the details of language features and library methods using the many sources of documentation available on the web.

Several of the methods in Figure 8-2 have a direct correspondence to methods in the String class. For example, the concat method corresponds to the + operator for strings, the indexOf and lastIndexOf are essentially the same for both classes, and the slice method is the array counterpart of substring. The methods that are different are the ones that change the elements in an array, which have no equivalent for strings.

The last two array methods shown in Figure 8-2 provide tools for common high-level operations. The reverse method, for example, provides a built-in version of the reverseArray function presented earlier in this chapter. Because

### FIGURE 8-2 Common operations in JavaScript’s Array class

<table>
<thead>
<tr>
<th>Array field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array.length</code></td>
<td>Returns the number of elements in <code>array</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array.concat(a1, ...)</code></td>
<td>Concatenates this array with any number of arrays and then returns a copy of the concatenated result.</td>
</tr>
<tr>
<td><code>array.indexOf(value, k)</code></td>
<td>Returns the first index at which <code>value</code> appears, or -1 if the value does not appear in the array. If <code>k</code> is specified, it indicates the starting point.</td>
</tr>
<tr>
<td><code>array.lastIndexOf(value, k)</code></td>
<td>Returns the last index at which <code>value</code> appears, or -1 if the value does not appear in the array. If <code>k</code> is specified, it indicates the starting point of the reverse search.</td>
</tr>
<tr>
<td><code>array.pop()</code></td>
<td>Removes and returns the last element, or undefined if empty.</td>
</tr>
<tr>
<td><code>array.push(value, ...)</code></td>
<td>Adds one or more values to the end of the array.</td>
</tr>
<tr>
<td><code>array.shift()</code></td>
<td>Removes and returns the first element, or undefined if empty.</td>
</tr>
<tr>
<td><code>array.unshift(value, ...)</code></td>
<td>Adds one or more values to the beginning of the array.</td>
</tr>
<tr>
<td><code>array.slice(start, finish)</code></td>
<td>Returns a new array containing all elements beginning at <code>start</code> and ending just before <code>finish</code>.</td>
</tr>
<tr>
<td><code>array.splice(index, count, ...)</code></td>
<td>Removes <code>count</code> elements starting at position <code>index</code> and then inserts any additional arguments at that index position.</td>
</tr>
<tr>
<td><code>array.reverse()</code></td>
<td>Reverses the elements of the array and then returns the array.</td>
</tr>
<tr>
<td><code>array.sort()</code></td>
<td>Sorts the elements of the array and then returns the array.</td>
</tr>
</tbody>
</table>
reverse is a method and reverseArray is a function, you apply these operations in different ways. To reverse an array using the reverseArray function, clients would call

reverseArray(array);

To achieve the same result using the library reverse method, clients would instead write

array.reverse();

Similarly, calling

array.sort();

reorders the elements of the array so that they appear in ascending order.

Of the remaining methods, the most useful one is push, which adds one or more elements to the end of an existing array. The existence of the push method makes it easy to expand an array one element at a time in a fashion similar to the pattern for growing a string by concatenation. For example, the push method makes it possible to write the following function, which creates an array of n elements, each of which is initialized to value:

function createArray(n, value) {
  let array = [ ];
  for (let i = 0; i < n; i++) {
    array.push(value);
  }
  return array;
}

For example, calling createArray(8, false) allocates an array with 8 elements, each of which is initialized to the Boolean value false:

<table>
<thead>
<tr>
<th>false</th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Accessing elements outside the array bounds

Arrays in JavaScript are implemented in a different way than arrays in most other programming languages. In fact, JavaScript arrays are implemented just like other compound objects. What makes arrays different is how programmers use them. When working with arrays, programmers conventionally use numeric indices of the sort you have already seen in this chapter. When working with objects, programmers use symbolic names to refer to individual components, as you will
discover in Chapter 9. Since JavaScript uses the same internal representation for both arrays and objects, it turns out that you can violate this convention without generating any errors from the JavaScript interpreter.

One of the implications of the shared underlying representation between arrays and objects is that JavaScript allows you to refer to elements in an array whose indices fall outside the range of the defined elements. If you ask for the value of such an element, JavaScript returns the value \textit{undefined}. If you assign a new value to an element outside the defined bounds, JavaScript creates that element, even if doing so leaves undefined elements interspersed among the defined ones.

Suppose, for example, that you have written the declaration

```javascript
let array = [1, 2, 3];
```

which gives rise to the following box diagram:

```
array
1 2 3
0 1 2
```

Given that array has only three elements, it seems reasonable that asking for the value of \texttt{array[7]} would return the value \textit{undefined}. But what would happen if you were to try to assign a value to that element, as in

```javascript
array[7] = 8;
```

When JavaScript executes this statement, it creates an element at index 7, leaving holes in the array at index positions 3, 4, 5, and 6, as follows:

```
array
1 2 3 8
0 1 2 3 4 5 6 7
```

JavaScript’s interpretation of arrays means that you can fill the contents of an array simply by assigning to its index positions. For example, you can rewrite the \texttt{createArray} function from the previous page like this:

```javascript
function createArray(n, value) {
    let array = [ ];
    for (let i = 0; i < n; i++) {
        array[i] = value;
    }
    return array;
}
```
8.2 Reading files

Although variables are wonderful for storing information during the lifetime of a program, the information in variables is ephemeral. When a program finishes, the values of its variables are lost. Whenever you want to store information more permanently, the usual approach is to collect the data into a file, which is a logically cohesive collection of data stored using some technology that allows the data to persist beyond the execution of a program. Each file is identified by a filename, which allows the computer to keep it separate from other files. The filename and the data contained in the file have the same relationship as the name of a variable and its contents.

Ordinarily, files are stored on a hard disk inside the machine, but may also be stored on removable medium, such as a flash drive. In either case, the basic principles and modes of operation remain the same. The important point is that the permanent data objects you store on the computer—documents, games, executable programs, source code, and the like—are all stored in the form of files.

Files come in a variety of types. On your computer, you probably have sound files, image files, video files, and a variety of other file types. As you know from the discussion of binary representation in Chapter 7, the contents of these files are represented internally as sequences of bits that are interpreted in different ways corresponding to the different types of data.

Text files

One of the most important file types is the text file, which contains a sequence of characters. In many ways, text files are analogous to strings. The most significant difference is that text files are stored in a permanent way outside the context of a particular program while strings are stored internally. Text files are usually created using an editor such as the one you’ve been using to create the various .js and .html files—all of which are text files—for the programs in this book.

Mostly to have some text files to use as examples, imagine that you want to collect a set of your favorite quotations from Shakespeare and have decided to store each quotation in a separate file. You might begin your collection with the following lines from Hamlet:

```
To be, or not to be: that is the question.
Whether ’tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them?
```
For a second quotation, you might choose the following lines from Juliet’s balcony scene in *Romeo and Juliet*:

![Image: Juliet.txt]

```
What's in a name?
That which we call a rose
By any other name would smell as sweet.
```

And because it will be useful in later illustrations to have a file that is a little shorter, you might also create the following file:

![Image: Macbeth.txt]

```
A drum, a drum!
Macbeth doth come.
```

Each of these diagrams shows the filename outside the box that encloses the contents of the file, just as diagrams of variables show the name on the outside and the value on the inside.

Internally, text files are represented as a continuous sequence of characters in which the individual lines are separated with a *newline character*, which marks the end of a line. For example, the contents of the file *Macbeth.txt* look like this:

![Image: Macbeth contents]

```
A drum, a drum!
Macbeth doth come.
```

Each line ends with the newline character, which is indicated in JavaScript strings using the two-character sequence `\n`.

**Reading files in JavaScript**

Unfortunately, files and JavaScript do not play well together. JavaScript is used primarily to implement interactive content that runs in a browser. For security reasons, JavaScript programs running in a browser are not allowed to read arbitrary files on the user’s computer. If that were possible, malicious web sites could use that ability to collect sensitive data from the user’s file system. The one exception to this rule is that JavaScript code running in a browser can read files if the user deliberately chooses a file in a dialog initiated in response to an explicit user command.

File operations in JavaScript are also complicated by the fact that reading the contents of a file is an operation that might take some time, depending on the size of the file. Most languages therefore include library functions that read data from a file and wait for that process to complete before proceeding. That model, however, is not appropriate for JavaScript, which doesn’t support the concept of suspending
execution while the library function waits for the read operation to complete. In JavaScript, reading files uses much the same model it requires for other events. If you want to read a file in JavaScript, you need to call a library function to start the operation, passing in a callback function that the library can then call when the read is complete.

The programs in this book use a library called JSFile.js, which defines a few functions to make the process of reading files as painless as possible. As with any other library, you need to include the JSFile.js library in a <script> tag that looks like this:

```
<script type="text/javascript" src="JSFile.js"></script>
```

The functions in the JSFile library are similar to those in the built-in Math library in the sense that they are invoked as functions belonging to a class rather than as methods on an object. In practice, this implementation strategy means that you need to write JSFile and a period before the function name, just as you include the class name when you call a function like Math.sqrt.

The most important function in the JSFile library is chooseTextFile, which allows the user to choose a file and then initiates the operation of reading that file. The JSFile.chooseTextFile function takes a callback function as an argument. When JavaScript has finished reading the file, it invokes the callback function passing it the entire contents of the file as a string.

The operation of JSFile.chooseTextFile is easiest to explain by example. Suppose that you want to write a simple program that lets the user select a file, which is then read and displayed on the console. A simple program that comes close to achieving this goal looks like this:

```javascript
function ListFile() {
    function displayContents(text) {
        console.log(text);
    }
    JSFile.chooseTextFile(displayContents);
}
```

The ListFile function begins by defining the callback function, which is called displayContents. This function takes a single parameter named text, which will be set to the entire contents of the file when the read operation is complete. The only other line in ListFile is the call to JSFile.chooseTextFile, which looks like this:

```
JSFile.chooseTextFile(displayContents);
```
The effect of this call is to pop up a small dialog button in the center of the browser window that asks the user to initiate the process of choosing a file. Clicking this button then brings up a file dialog in the window that lets the user navigate through the local file system and select a file. The appearance of the file dialog depends on the operating system but will typically look exactly like the standard browser you see on your computer. Here, for example, is a screen image of the file browser on my Macintosh laptop after going through the various folders to find the one that contains the files for this chapter:

Clicking on `Macbeth.txt` enables the **Open** button, like this:

Clicking **Open** asks the browser to open the selected file, which is `Macbeth.txt`. When the browser has finished reading the file, it combines the text of the file into a single string and passes that to the callback function `displayFile`. That function
then calls `console.log` on the contents of the file, which generates the following console output:

```
<table>
<thead>
<tr>
<th>ListFile</th>
</tr>
</thead>
<tbody>
<tr>
<td>A drum, a drum!</td>
</tr>
<tr>
<td>Macbeth doth come.</td>
</tr>
</tbody>
</table>
```

This output looks precisely as it should, but the code for `ListFile` is marked with a bug symbol. The problem is that there is an extraneous blank line at the end of the output. The file contains a newline character at the end of the last line, which therefore appears on the console when the contents of the file are displayed in the call to `console.log`. The `console.log` function, however, always adds a newline to the end of its output, thereby creating an empty line.

### Reading files as an array of lines

The easiest way to fix the problem is to use the `split` method in the string library to convert the string containing the entire file into an array of individual lines. The `split` method takes a separator string, divides the string at each instance of that separator, and then returns an array of the strings marked by those divisions. For example, if the variable `str` contains the string "16-Jul-1969" (the date of the Apollo 11 moon landing), calling `str.split("-")` would return the following array:

```
"16"  "Jul"  "1969"
```

You can use `split` in a similar way to divide a text string into its component lines by specifying the string "\n" as the separator, but the fact that a text file usually ends with a newline introduces a small complication. If you call `split` on the contents of `Macbeth.txt`, you get the following three-element array:

```
"A drum, a drum!"  "Macbeth doth come."  ""
```

The empty string at the end appears because `split` interprets its argument as a separator character and therefore includes the text after the last newline character in the array. This problem is analogous to the fencepost problem introduced in Chapter 1: if there are two instances of a separator character in a string, those separators divide the string into three pieces. If you want instead to produce an array that matches the conceptual interpretation of the file as an array of two lines, you need to remove the empty string at the end of the array, as illustrated by the following function:
function convertToLineArray(text) {
    let lines = text.split("\n");
    if (lines.length > 0 &&
        lines[lines.length - 1] === "") {
        lines.pop();
    }
    return lines;
}

This implementation correctly handles the case when the last line of the file does not include a newline. The `convertToLineArray` function is so useful in reading files that it is included in the `JSFile` library.

Once you have converted the contents of a file to an array of lines, you can go through the lines of the file by applying the appropriate array operations. You can, for example, use a `for` loop to cycle through each of the lines. In many applications, however, it is more convenient to use the `shift` method to obtain the next line of the file and at the same time remove it from the array using the following pattern:

```javascript
while (lines.length > 0) {
    let line = lines.shift();
    . . . code to process the current line . . .
}
```

This pattern is illustrated in Figure 8-3, which correctly implements the `ListFile` function.

---

**FIGURE 8-3** Program to list the contents of a file chosen by the user

```javascript
/*
 * File: ListFile.js
 * ------------------
 * This program allows the user to choose a file and then lists the
 * contents of that file on the console.
 */

function ListFile() {
    function displayContents(text) {
        let lines = JSFile.convertToLineArray(text);
        while (lines.length > 0) {
            console.log(lines.shift());
        }
    }
    JSFile.chooseTextFile(displayContents);
}
8.3 Using arrays for tabulation

The data structure of a program is typically designed to reflect the organization of data in the real-world domain of the application. In general, whenever an application involves data that can be represented in the form of a list with elements $a_0, a_1, a_2$, and so on, an array is the natural choice for the underlying representation. It is also quite common for programmers to refer to the index of an array element as a subscript, reflecting the fact that arrays are used to hold data that would typically be written with subscripts in mathematics.

There are, however, important uses of arrays in which a different relationship exists between the data in the application domain and the data in the program. Instead of storing the data values in successive elements of an array, it makes more sense for some applications to use the data to generate array indices. Those indices are then used to select elements in an array that record some statistical property of the data as a whole. Understanding how this approach works and appreciating how it differs from more traditional uses of arrays requires looking at a concrete example, such as the letter-frequency application described in the following section.

Counting letter frequencies

The exercises for Chapter 7 ask you to write a program that implements a letter-substitution cipher, which encrypts a message by replacing each letter in the input text with an encoded version of that letter determined through the use of a secret key. Although the task of implementing a letter-substitution cipher is an interesting problem in its own right, an even more interesting computational problem is figuring out how you might break a letter-substitution cipher if you did not have access to the key.

The problem of breaking a letter-substitution cipher is so straightforward that it often appears in recreational puzzles called cryptograms. Edgar Allan Poe was a great fan of cryptograms and described a technique for solving them in his 1843 novel *The Gold Bug*:

My first step was to ascertain the predominant letters, as well as the least frequent. Counting all, I constructed a table... Now, in English, the letter which most frequently occurs is e. Afterwards, the succession runs thus: a o i d h n r s t u y c f g l m w b k p q x z. E however predominates so remarkably that an individual sentence of any length is rarely seen, in which it is not the prevailing character.

As Poe suggests, the first step in discovering the hidden meaning of a cryptogram is to construct a table showing how often each letter appears. A program that does just that appears in Figure 8-4.
Arrays

Figure 8-4 Program to count letter frequencies in a file

/*
 * File: CountLetterFrequencies.js
 * -------------------------
 * This function displays a table showing the letter frequencies in a file.
 */

function CountLetterFrequencies() {
    let processContents = function(text) {
        displayFrequencyTable(createFrequencyTable(text));
    }
    JSFile.chooseTextFile(processContents);
}

/*
 * Creates a letter frequency table from the specified string.
 */

function createFrequencyTable(str) {
    let letterCounts = createArray(26, 0);
    for (let i = 0; i < str.length; i++) {
        let ch = str.charAt(i).toUpperCase();
        if (ch >= "A" && ch <= "Z") {
            letterCounts[ch.charCodeAt(0) - "A".charCodeAt(0)]++;
        }
    }
    return letterCounts;
}

/*
 * Displays the frequency table. Letters that never appear are not shown.
 */

function displayFrequencyTable(letterCounts) {
    for (let i = 0; i < 26; i++) {
        let count = letterCounts[i];
        if (count > 0) {
            let ch = String.fromCharCode("A".charCodeAt(0) + i);
            console.log(ch + ": " + count);
        }
    }
}

/*
 * Creates an array containing n elements, each of which is set to value.
 */

function createArray(n, value) {
    let array = [];
    for (let i = 0; i < n; i++) {
        array.push(value);
    }
    return array;
}
As it happens, Poe’s list of the most common letters is not in fact correct. Computerized analysis reveals that the 12 most common letters in English are

\[
E \ T \ A \ O \ I \ N \ S \ H \ R \ D \ L \ U
\]

Given that computerized analyses of English text were not available in his day, Poe can perhaps be excused for making a few mistakes.

The `CountLetterFrequencies.js` program in Figure 8-4 begins by calling `JSFile.chooseTextFile`, which pops up a dialog that allows the user to select the input file. The callback function `processContents` passes the text of the file to the function `createFrequencyTable` to count how often each of the 26 letters appears. When `createFrequencyTable` returns, its result is passed immediately to the function `displayFrequencyTable`, which displays the letter frequencies on the console.

Suppose, for example, that you have created a data file named `Suess.txt` containing the first page from one of Dr. Suess’s popular “I can read it all by myself” books:

<table>
<thead>
<tr>
<th>Suess.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>One fish</td>
</tr>
<tr>
<td>Two fish</td>
</tr>
<tr>
<td>Red fish</td>
</tr>
<tr>
<td>Blue fish</td>
</tr>
</tbody>
</table>

If you then run `CountLetterFrequencies.js` and select the file `Suess.txt`, the program produces the following display on the console:

<table>
<thead>
<tr>
<th>CountLetterFrequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>B: 1</td>
</tr>
<tr>
<td>D: 1</td>
</tr>
<tr>
<td>E: 3</td>
</tr>
<tr>
<td>F: 4</td>
</tr>
<tr>
<td>H: 4</td>
</tr>
<tr>
<td>I: 4</td>
</tr>
<tr>
<td>L: 1</td>
</tr>
<tr>
<td>N: 1</td>
</tr>
<tr>
<td>O: 2</td>
</tr>
<tr>
<td>R: 1</td>
</tr>
<tr>
<td>S: 4</td>
</tr>
<tr>
<td>T: 1</td>
</tr>
<tr>
<td>U: 1</td>
</tr>
<tr>
<td>W: 1</td>
</tr>
</tbody>
</table>

The output shows that the file contains four copies of the letters \(F, I, S, \text{ and } H\) (one for each of the four appearances of \textit{fish}), three \(E’s\), two \(O’s\), and a smattering of letters that each appear exactly once. Note that letters that don’t appear in the file are not shown in the output.
The `CountLetterFrequencies.js` can work with arbitrarily large files. Running this program on a file containing the full text of George Eliot’s *Middlemarch* produces the following output:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>114157</td>
</tr>
<tr>
<td>B</td>
<td>23269</td>
</tr>
<tr>
<td>C</td>
<td>34031</td>
</tr>
<tr>
<td>D</td>
<td>61046</td>
</tr>
<tr>
<td>E</td>
<td>166999</td>
</tr>
<tr>
<td>F</td>
<td>30826</td>
</tr>
<tr>
<td>G</td>
<td>30055</td>
</tr>
<tr>
<td>H</td>
<td>89636</td>
</tr>
<tr>
<td>I</td>
<td>99651</td>
</tr>
<tr>
<td>J</td>
<td>1695</td>
</tr>
<tr>
<td>K</td>
<td>11010</td>
</tr>
<tr>
<td>L</td>
<td>56865</td>
</tr>
<tr>
<td>M</td>
<td>37816</td>
</tr>
<tr>
<td>N</td>
<td>96887</td>
</tr>
<tr>
<td>O</td>
<td>108561</td>
</tr>
<tr>
<td>P</td>
<td>21922</td>
</tr>
<tr>
<td>Q</td>
<td>1441</td>
</tr>
<tr>
<td>R</td>
<td>79908</td>
</tr>
<tr>
<td>S</td>
<td>88555</td>
</tr>
<tr>
<td>T</td>
<td>123433</td>
</tr>
<tr>
<td>U</td>
<td>40647</td>
</tr>
<tr>
<td>V</td>
<td>12792</td>
</tr>
<tr>
<td>W</td>
<td>34508</td>
</tr>
<tr>
<td>X</td>
<td>2069</td>
</tr>
<tr>
<td>Y</td>
<td>28700</td>
</tr>
<tr>
<td>Z</td>
<td>249</td>
</tr>
</tbody>
</table>

If you sort this output by letter frequency in descending order, you discover that the 12 most common letters in *Middlemarch* are:

\[
E \quad T \quad A \quad O \quad I \quad N \quad H \quad S \quad R \quad D \quad L \quad U
\]

The only difference between this frequency table and the statistical results for modern English presented on page 273 is that the *H* and the *S* are reversed, which may reflect nothing more than the fact that the name of the central character, Dorothea, contains the letter *H*. In general, the more text you analyze, the closer the frequencies will come to those calculated for modern English.

The strategy used in the `CountLetterFrequencies.js` program is to create an array of 26 integers and then to use the character code for the letters in the file to select the appropriate element within the array. Each element in the array contains an integer representing the current count of the letter that corresponds to that index. Thus, the element at the beginning of the array corresponds to the number of *A*’s, and the element at the end of the array corresponds to the number of *Z*’s. If you call the array `letterCounts`, you can initialize it by writing

```
let letterCounts = createArray(26, 0)
```
This declaration uses the `createArray` function defined earlier in this chapter to allocate space for an array with 26 elements, as shown in this diagram:

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |15 |16 |17 |18 |19 |20 |21 |22 |23 |24 |25 |
```

Each time a letter appears in the input, you need to increment the corresponding element in `letterCounts`. Finding the element to increment is simply a matter of converting the character into an integer in the range 0 to 25 by using converting the character to uppercase and then subtracting the Unicode value of "A". If the character from the file is stored in the variable `ch`, the code necessary to increment the appropriate element in `letterCounts` looks like this:

```
letterCounts[ch.charCodeAt(0) - "A".charCodeAt(0)]++;
```

The code for `displayFrequencyTable` has to perform the same conversion in the opposite direction. The values of `i` in the `for` loop run from 0 to 25. To convert that integer to a character requires the following code:

```
let ch = String.fromCharCode("A".charCodeAt(0) + i);
```

### 8.4 Multidimensional arrays

In JavaScript, the elements of an array can be of any type. In particular, the elements of an array can themselves be arrays. Arrays of arrays are called **multidimensional arrays**. The most common form of multidimensional array is the two-dimensional array, which is most often used to represent data in which the individual entries form a rectangular structure marked off into rows and columns. This type of two-dimensional structure is called a **matrix**. Arrays of three or more dimensions are also legal in JavaScript but occur much less frequently.

As an example of a two-dimensional array, suppose you wanted to represent a game of Tic-Tac-Toe as part of a program. As you probably know, Tic-Tac-Toe is played on a board consisting of three rows and three columns, as follows:

```
  
  
  
```

Players take turns placing the letters X and O in the empty squares, trying to line up three identical symbols horizontally, vertically, or diagonally.
To represent the Tic-Tac-Toe board, the most natural strategy is to use a two-dimensional array with three rows and three columns. The simplest choice for the individual elements is to make them strings, which must be one of the following: "" (representing an empty square), "X", and "O". Since the board is initially empty, the declaration can look like this:

```javascript
let board = [ [ "", "", "" ],
              [ "", "", "" ],
              [ "", "", "" ] ];
```

Given this declaration, you could then refer to the characters in the individual squares by supplying two separate indices, one specifying the row number and another specifying the column number. In this representation, each number varies over the range 0 to 2, and the individual positions in the board have the following designations:

<table>
<thead>
<tr>
<th>board[0][0]</th>
<th>board[0][1]</th>
<th>board[0][2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>board[1][0]</td>
<td>board[1][1]</td>
<td>board[1][2]</td>
</tr>
</tbody>
</table>

Internally, JavaScript represents the variable `board` as an array of three elements, each of which is an array of three strings. By convention, the elements in a multidimensional array are arranged so that the first index value varies less rapidly than the second, and so on. Thus all the elements of `board[0]` appear before any elements of `board[1]`. Because this strategy goes through all the elements on each row before it proceeds to the next one, this ordering is called row-major order.

### 8.5 Image processing

In modern computing, one of the most important applications of two-dimensional arrays occurs in the field of computer graphics. As you learned in Chapter 3, graphical images are composed of individual pixels; Figure 3-6 on page 74 offers a magnified view of the screen that shows how the pixels create the image as a whole. The connection with this chapter is that the pixels are arranged so that they form a two-dimensional array.

**The GImage class**

The Stanford Graphics Library defines the `GImage` class as a graphical object that contains image data encoded using one of the standard formats. The three most
common are the Portable Network Graphics (PNG) format, the Joint Photographic Experts Group (JPEG) format, and the Graphics Interchange Format (GIF). Although most browsers are capable of displaying images encoded in other formats as well, you can maximize the portability of your program by sticking with the most common formats. The image files distributed with this book appear in PNG format.

The first step in displaying an image is to create or download an image file in one of the standard formats. The name of the image file should end with an extension that identifies the encoding format, which is typically .png. You then need to store that file on your computer in the same directory as the index.html file. In your program, you then create a GImage object and add it to the graphics window, just as you would with any of the other GObject subclasses. For example, if you have an image file called MyImage.png, you can display that image in the upper left corner of the graphics window using the following line:

```javascript
  gw.add(GImage("MyImage.png"));
```

**Determining properties of an image**

Unfortunately, the situation is not quite so simple if you need to use properties of an image to determine where the image should be placed. Suppose, for example, you wanted to center the image in the graphics window. Since GImage is a subclass of GObject, it implements the methods getWidth and getHeight, which suggests that you could center the image using the following code:

```javascript
  let image = GImage("MyImage.png");
  let x = (gw.getWidth() - image.getWidth()) / 2;
  let y = (gw.getHeight() - image.getHeight()) / 2;
  gw.add(image, x, y);
```

The problem with this code segment is that JavaScript implements reading an image as an asynchronous operation. Calling the GImage function starts the process of reading the image but does not wait for that process to complete. The implementation of the GImage class knows how to update the image on the graphics window when the image is fully loaded, but it is impossible to get any information about the image until that process is complete. In particular, you can’t determine the size of the image by calling getWidth and getHeight. Since you need this information to center the image, you have to make sure that the image is fully loaded before you can determine where to place it in the window.

Like the GWindow class, the GImage class implements the addEventListener method, which takes the name of an event and a callback function that is called when the event occurs. For GImage, the relevant event is the "load" event, which
is triggered when the event is fully loaded. The `addEventListener` method makes it possible to center an image using the following code:

```javascript
let image = GImage("MyImage.png");

function displayImage() {
    let x = (gw.getWidth() - image.getWidth()) / 2;
    let y = (gw.getHeight() - image.getHeight()) / 2;
    gw.add(image, x, y);
}

image.addEventListener("load", displayImage);
```

You also need to use a callback function if you need to scale an image so that it has the desired size. The code for the `EarthImage.js` program in Figure 8-5 illustrates the use of scaling to display an image so that it fills the available space in the window. The image, which shows the earth as seen by the Apollo 17 astronauts

---

**Figure 8-5** Program to draw an image of the earth taken from Apollo 17

```javascript
/*
* File: EarthImage.js
* -----------------
* This program draws a picture of the earth taken by Apollo 17 along with
* a photo credit.
*/

/* Constants */
const GWINDOW_WIDTH = 400;
const GWINDOW_HEIGHT = 415;
const CITATION_FONT = "12px 'Helvetica Neue';";
const CITATION_Y = 3;

/* Main program */
function EarthImage() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let image = GImage("EarthImage.png");

    function addImageToWindow() {
        image.scale(gw.getWidth() / image.getWidth());
        gw.add(image, 0, 0);
    }

    image.addEventListener("load", addImageToWindow);
    let citation = GLabel("Courtesy NASA/JPL-Caltech ");
    citation.setFont(CITATION_FONT);
    let x = gw.getWidth() - citation.getWidth();
    let y = gw.getHeight() - CITATION_Y;
    gw.add(citation, x, y);
}
```
on their way to the moon in December 1972, is stored in an image file named EarthImage.png. The EarthImage.js program reads that image file into a GImage object and the callback function then adds that object to the window.

The EarthImage.js program introduces one new method from the GObject class, which is particularly useful for images. The scale method changes the size of the image by the specified scale factor, which is either a single number applied to both dimensions or a pair of numbers applied independently to the x and y axes. If the variable image contains a GImage object, calling

```javascript
image.scale(0.5);
```

resizes the object so that it is half as big in each dimension. Calling

```javascript
image.scale(2, 1);
```

doubles the size of the image horizontally but leaves the vertical size unchanged. In this case, calling

```javascript
image.scale(gw.getWidth() / image.getWidth());
```

scales the image uniformly in the x and y axes so that it fills the entire width of the window. This calculation can be performed only in the callback function because it needs to know the actual width of the image. Loading the index.html file for the EarthImage.js program produces the following image in the browser window:
The EarthImage.js program also illustrates the inclusion of a citation along with an image. When you use existing images, you need to be aware of possible restrictions on the use of intellectual property. Most of the images you find on the web are protected by copyright. Under copyright law, you must obtain the permission of the copyright holder in order to use the image, unless your use of the image satisfies the guidelines for “fair use”—a doctrine that has unfortunately become much more murky in the digital age. Under “fair use” guidelines, you could almost certainly use a copyrighted image in a paper that you write for a class. On the other hand, you could not put that same image into a commercially published work without first securing—and probably paying for—the right to do so.

Even in cases in which your use of an image falls within the “fair use” guidelines, it is important to give proper credit to the source. As a general rule, whenever you find an image on the web that you would like to use, you should first check to see whether that web site explains its usage policy. Many of the best sources for images on the web have explicit guidelines for using their images. Some images are absolutely free, some are free for use with citation, some can be used in certain contexts but not others, and some are completely restricted. For example, the web site for the National Aeronautics and Space Administration (http://www.nasa.gov) has an extensive library of images about the exploration of space. As the web site explains, you can use these images freely as long as you include the citation “Courtesy NASA/JPL-Caltech” along with the image. The EarthImage.js program follows these guidelines and includes the requested citation on the page with the image.

### Representation of images

In JavaScript, an image is a rectangular array in which the image as a whole is a sequence of rows, and each row is a sequence of individual pixel values. The value of each element in the array indicates the color that should appear in the corresponding pixel position on the screen. From Chapter 3, you know that you can specify a color in JavaScript by indicating the intensity of each of the primary colors. Each of those intensities ranges from 0 to 255 and therefore fits in an eight-bit byte. The color as a whole is stored in a 32-bit integer that contains the red, green, and blue intensity values along with a measure of the transparency of the color, represented by the Greek letter alpha (α). For the opaque colors used in most images, the value of α is always 255 in decimal, which is 11111111 in binary or FF in hexadecimal.

As an example, the following diagram shows the four bytes that form the color PINK, which JavaScript defines using the hexadecimal values FF, C0, and CB as the red, green, and blue components. Translating those values to their binary form gives you the following:
The fact that JavaScript packs all the information about a color into a 32-bit integer means that you can store an image as a two-dimensional array of integers. At the outer level, each element of the array is an entire row of the image. In keeping with JavaScript’s coordinate system, the rows of an image are numbered from 0 starting at the top of the image. Each row is an array of integers representing the value of each pixel as you move from left to right across the image.

### Using the GImage class to manipulate images

The GImage class in the graphics library exports several methods that make it possible to perform basic image processing. As long as certain conditions are met concerning the source of the image, you can obtain the two-dimensional array of pixel values by calling getPixelArray. Thus, if the variable image contains a GImage, you can retrieve its pixel array by calling

```
let pixelArray = image.getPixelArray();
```

The height of the image is equal to the number of rows in the pixel array. The width is the number of elements in any of the rows, each of which has the same length in a rectangular image. Thus, you can initialize variables to hold the height and width of the pixel array like this:

```
let height = pixelArray.length;
let width = pixelArray[0].length;
```

Unfortunately, many browsers do not allow JavaScript programs to obtain the pixel array from an arbitrary image. As was true for text files, the security features implemented as part of the browser prohibit JavaScript code from reading the contents of an image unless the user explicitly selects that image. To make it possible to write programs that manipulate images, the JSFile library includes a method called chooseDataFile that operates similarly to chooseTextFile except that the file contains data in some binary encoding other than text. The callback function passed to JSFile.chooseDataFile takes a single argument containing the contents of the data file in a web-compatible format called a data URL, which is a text-based representation of the binary contents. The GImage function accepts a data URL as an argument and creates an image from the stored data.

The ChooseImage.js program in Figure 8-6 at the top of the next page illustrates the use of the JSFile.chooseDataFile function to choose an image file. In this program, the createImage callback function creates the image from
the data URL and then registers the function `displayImage` as a second level of callback processing that occurs once the image is fully loaded. The `displayImage` function can then call `getWidth` and `getHeight` on the complete image to determine where it should be placed to center the image in the window.

With its two levels of callback processing, the `ChooseImage.js` program gives you a sense that JavaScript's strategy of requiring callback functions to process data after an event occurs can get rather complicated. Most languages avoid this complexity by allowing this processing to occur synchronously. In those languages, your program would call a function to choose an image, and the process would then wait until that image was selected. Once the selection was complete, the process would again wait for the image to be loaded. Synchronous models turn out to be much easier for the programmer, particularly as the number of steps becomes larger.
Using synchronous models in the context of a browser, however, is problematic because the browser typically becomes unresponsive when the process is waiting for an event or the completion of a long-running process. To avoid this problem, JavaScript uses an asynchronous model based on callback functions as described in this chapter.

If the program logic becomes much more complicated than the examples in this chapter, keeping track of the series of callback functions that need to be executed can create a conceptual nightmare for the programmer. To make this process more manageable, modern versions of JavaScript include a feature called *promises*, which are a kind of data structure that keeps track of operations that can happen at some time in the future. Although promises are beyond the scope of this text, you should look up how to use them if you find yourself writing complex asynchronous code.

The `GImage` class includes several methods to simplify the task of manipulating image data. These methods appear in Figure 8-7. As you can see from the first section of the figure, the `GImage` class supports two factory methods, one for reading data from a file and one to construct a `GImage` from a two-dimensional

![Figure 8-7 Useful methods in the GImage class](image)

### Factory methods to create a GImage

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GImage(filename)</code></td>
<td>Creates a new <code>GImage</code> by reading the image data from the specified file.</td>
</tr>
<tr>
<td><code>GImage(array)</code></td>
<td>Creates a new <code>GImage</code> from the pixel array.</td>
</tr>
</tbody>
</table>

### Method to specify a callback function

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>image.addEventListener(&quot;load&quot;, fn)</code></td>
<td>Adds a callback function to the image that is called when the image is fully loaded.</td>
</tr>
</tbody>
</table>

### Method to read the individual pixels in an image

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>image.getPixelArray()</code></td>
<td>Returns the pixel array for this image.</td>
</tr>
</tbody>
</table>

### Class methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GImage.getRed(pixel)</code></td>
<td>Returns the red component of the pixel as an integer between 0 and 255.</td>
</tr>
<tr>
<td><code>GImage.getGreen(pixel)</code></td>
<td>Returns the green component of the pixel as an integer between 0 and 255.</td>
</tr>
<tr>
<td><code>GImage.getBlue(pixel)</code></td>
<td>Returns the blue component of the pixel as an integer between 0 and 255.</td>
</tr>
<tr>
<td><code>GImage.createRGBPixel(r, g, b)</code></td>
<td>Creates a pixel value from the specified <em>r</em>, <em>g</em>, and <em>b</em> components, each of which is between 0 and 255.</td>
</tr>
<tr>
<td><code>GImage.createPixelArray(width, height)</code></td>
<td>Creates a pixel array of the specified size.</td>
</tr>
</tbody>
</table>
Given an initialized image, the `getPixelArray` method returns the array of pixels stored within the image. The `GImage` class also exports methods for retrieving the red, green, and blue components of a pixel expressed as an integer and for assembling independent red, green, and blue values into the corresponding integer form.

These new capabilities in the `GImage` class make it possible for you to write programs to manipulate images, in much the same way that a commercial system like Adobe Photoshop™ does. The general strategy consists of the following three steps, all of which must be performed in the callback function to ensure that the image is fully loaded:

1. Use `getPixelArray` to obtain the array of pixel values.
2. Perform the desired transformation by manipulating the values in the array.
3. Call the `GImage` function to create a new object from the modified array.

The following function definition uses this pattern to flip an image vertically:

```javascript
function flipVertical(image) {
  let array = image.getPixelArray();
  array.reverse();
  return new GImage(array);
}
```

In this example, you don’t actually have to look at the individual pixels. All you have to do is reverse the order of the rows.

A more substantive problem is that of converting a color image to grayscale, a format in which all the pixels are either black, white, or some intermediate shade of gray. To do so, you need to go through each element in the pixel array and replace each pixel with a new shade of gray that approximates the apparent brightness of that color. In computer graphics, that apparent brightness is called luminance.

The goal of a grayscale conversion is to produce a shade of gray that approximates the brightness of each pixel to the eye. As it turns out, luminance does not depend on the color components equally and is controlled much more strongly by how much green appears in the pixel than by the amount of red or blue. Red and blue tend to make an image appear darker, while green tends to lighten things up. The formula for luminance adopted by the standards committee responsible for television signals in the United States looks like this:

\[ \text{luminance} = 0.299 \times \text{red} + 0.587 \times \text{green} + 0.114 \times \text{blue} \]

The complete code for the `GrayscaleImage.js` program appears in Figure 8-8.
8.5 Image processing 285

**Figure 8-8** Program to convert an image to grayscale

```javascript
/*
* File: GrayscaleImage.js
* ---------------------------------
* This file displays an image together with its grayscale equivalent.
*/

/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 400;
const IMAGE_SEP = 50;

/* Main program */

function GrayscaleImage() {
    let gw = newWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);

    function createImage(data) {
        let image = GImage(data);
        function displayImages() {
            gw.add(image, (gw.getWidth() - IMAGE_SEP) / 2 - image.getWidth(),
                    (gw.getHeight() - image.getHeight()) / 2);
            let grayscale = createGrayscaleImage(image);
            gw.add(grayscale, (gw.getWidth() + IMAGE_SEP) / 2,
                   (gw.getHeight() - image.getHeight()) / 2);
        }
        image.addEventListener("load", displayImages);
    }
    JFile.chooseDataFile(createImage);
}

/* Creates a grayscale image based on the luminance of each pixel */

function createGrayscaleImage(image) {
    let array = image.getPixelArray();
    let height = array.length;
    let width = array[0].length;
    for (let i = 0; i < height; i++) {
        for (let j = 0; j < width; j++) {
            let gray = luminance(array[i][j]);
            array[i][j] = GImage.createRGBPixel(gray, gray, gray);
        }
    }
    return GImage(array);
}

/* Returns the luminance of a pixel using the NTSC formula */

function luminance(pixel) {
    let r = GImage.getRed(pixel);
    let g = GImage.getGreen(pixel);
    let b = GImage.getBlue(pixel);
    return Math.round(0.299 * r + 0.587 * g + 0.114 * b);
}
```
As in the general pattern for image manipulation, the main program allows the user to choose an image and then waits to be sure that the image is loaded. The `displayImages` callback function then displays the original and grayscale images side by side, like this:

![Grayscale Image](image)

**Summary**

In this chapter, you have learned how to use *arrays*, which are the primary data structure that JavaScript uses to represent lists of data. The important points introduced in this chapter include:

- Like most languages, JavaScript includes a built-in *array* type for storing an ordered collection of elements. Each element in an array has an integer index that indicates its position in the array. Index numbers begin with 0.
- JavaScript arrays are most often created by enclosing a list of the elements in square brackets, separated by commas.
- The number of elements in a JavaScript array is stored in a field called `length`.
- You can select a particular element of an array by indicating the index of the desired element in square brackets after the array name. This operation is called `selection`.
- Arrays are often used together with *for* loops that allow you to cycle through the elements of the array.
- Arrays in JavaScript are stored as *references* to the memory containing the values of the array. An important implication of this design is that passing an array as a parameter does not copy the elements. Instead, JavaScript copies the reference value, which specifies the address of the array data. As a result, if a function changes the values of any elements of an array passed as a parameter, those changes will be visible to the caller.
- Arrays support a variety of operations implemented as methods. The most important array methods are listed in Figure 8-2 on page 262.
• Although JavaScript offers little support for data files, the code supplied with this book includes a `JSFile` library that exports methods that allow the user to select a file.

• JavaScript supports arrays with any number of dimensions, which are represented as arrays of arrays.

• Images are represented as two-dimensional arrays of integers, each of which specifies the color of a pixel as a combination of its red, green, and blue color components.

• The Stanford Graphics Library includes a class called `GImage` that supports images in a way that gives clients access to the underlying pixel array.

---

**Review questions**

1. Define the following terms as they apply to arrays: *element, index, length, and selection.*

2. How would you create an array called `dwarves` containing the names of the 13 dwarves who arrived at Bilbo’s doorstep in J. R. R. Tolkien’s fantasy, *The Hobbit*. Their names, in the order in which they appeared, are Dwalin, Balin, Kili, Fili, Dori, Nori, Ori, Oin, Gloin, Bifur, Bofur, Bombur, and Thorin.

3. How do you determine the length of an array?

4. True or false: Arrays violate the following rule for parameter passing as expressed in the following sentence from Chapter 5: The value of each argument is copied into the corresponding parameter variable.

5. What is a *reference*?

6. What is a *file*?

7. Why does JavaScript include such limited support for reading data files?

8. What is a multidimensional array?

9. The text suggests initializing the empty Tic-Tac-Toe board using the statement

   ```javascript
   let board = [ [ "", "", "" ],
                 [ "", "", "" ],
                 [ "", "", "" ] ];
   ```

   Would the following declaration accomplish the same task?

   ```javascript
   let board = createArray(3, createArray(3, ""));
   ```
Think carefully about how arrays are represented as references before offering your final answer.

10. What is row-major order?

11. What GObject subclass makes it possible to display images on the graphics window?

12. Describe how images are represented internally?

13. How do you extract the pixel array from an image?

14. Given a pixel array, how do you determine the width and height of the image?

15. In your own words, explain why each of the two callback functions is necessary in the ChooseImage.js program in Figure 8-6 or, equivalently, in the GrayscaleImage.js program in Figure 8-8.

**Exercises**

1. In statistics, a collection of data values is usually referred to as a *distribution*. A primary purpose of statistical analysis is to find ways to compress the complete set of data into summary statistics that express properties of the distribution as a whole. The most common statistical measure is the *mean* (traditionally denoted by the Greek letter $\mu$), which is simply the traditional average. Another common statistical measure is the *standard deviation* (traditionally denoted as $\sigma$), which provides an indication of how much the values in a distribution $x_1, x_2, \ldots, x_n$ differ from the mean. If you are computing the standard deviation of a complete distribution as opposed to a sample, the standard deviation can be expressed as follows:

$$
\sigma = \sqrt{\frac{\sum_{i=1}^{n} (\mu - x_i)^2}{n}}
$$

The uppercase sigma ($\Sigma$) indicates a summation of the quantity that follows, which in this case is the square of the difference between the mean and each individual data point.

Create a source library called StatsLib.js that exports the functions `mean` and `stdev`, each of which takes an array of numbers representing a distribution and returns the corresponding statistical measure. Make sure that your library includes enough commentary so that clients can understand how to use these functions.
2. Implement a function `createIndexArray(n)` that returns an array of `n` integers, each of which is set to its index in the array. For example, calling `createIndexArray(10)` should return the array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

3. Both the `reverseArray` function defined in this chapter and the `reverse` method that is defined for all array objects change the values in the calling array. An alternative approach, which was adopted for the `reverse` function for strings in Chapter 7, is to leave the parameter value unchanged and to instead return an entirely new value on which the operation has been applied. Use this strategy to implement a function `createReversedArray(array)` that returns an array whose elements are the reverse of the original values without changing the contents of `array`. Your function should allow you to generate the following console log:

```
> let hogwarts = [
    "Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"
];
> hogwarts
[Gryffindor, Hufflepuff, Ravenclaw, Slytherin]
> createReversedArray(hogwarts)
[Slytherin, Ravenclaw, Hufflepuff, Gryffindor]
> hogwarts
[Gryffindor, Hufflepuff, Ravenclaw, Slytherin]
```

4. A histogram is a graph that displays a set of values by dividing the data into separate ranges and then indicating how many data values fall into each range. For example, given the set of exam scores

100, 95, 47, 88, 86, 92, 75, 89, 81, 70, 55, 80

a traditional histogram would have the following form:

```
00s | 10s | 20s | 30s | 40s | 50s | 60s | 70s | 80s | 90s | 100
*   |     |     |     |     | *  |     | *   |     |     | 
```

The asterisks in the histogram indicate one score in the 40s, one in the 50s, five in the 80s, and so forth. When you generate histograms on the console, however, it is easier to display them sideways on the page, like this:
Write a program called Histogram that allows the user to select a file containing exam scores ranging from 0 to 100 and then displays a histogram of those scores, divided into the ranges 0–9, 10–19, 20–29, and so forth, up to the range containing only the value 100. Your function should match the format shown in the sample run as closely as you can.

6. Write a program that prints the longest line in a file chosen by the user. If several lines are all equally long, your program should print the first such line.

7. Write a program that allows the user to select a file and then reports how many lines, words, and characters appear in that file. For the purposes of this program, a word consists of a consecutive sequence of any characters except spaces and the end-of-line character. As an example, suppose that the file Lear.txt contains the following passage from Shakespeare’s King Lear:

```
Lear.txt
Poor naked wretches, wheresoe'er you are,
That bide the pelting of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loop'd and window'd raggedness, defend you
From seasons such as these? O, I have ta'en
Too little care of this!
```

If you run the program and select Lear.txt, your program should generate the following console output:

```
JavaScript Console
Chars: 254
Words: 43
Lines: 6
```  

8. A magic square is a two-dimensional array of integers in which the rows, columns, and diagonals all add up to the same value. One of the most famous magic squares appears in the 1514 engraving Melencolia I by Albrecht Dürer shown in Figure 8-9, in which a 4×4 magic square appears at the upper right,
just under the bell. In Dürer’s square, which can be read more easily in the magnified inset shown at the right of the figure, all four rows, all four columns, and both diagonals add up to 34. A more familiar example is the following 3×3 magic square in which each of the rows, columns, and diagonals add up to 15, as shown:
Implement a predicate function

```javascript
function isMagicSquare(square)
```

that tests to see whether `square` is a magic square. Your function should work for matrices of any size. If you call `isMagicSquare` with an array with a different number of rows and columns, your function should return `false`.

9. In the game of Minesweeper, a player searches for hidden mines on a rectangular grid that might—for a very small board—look like this:

![Minesweeper grid]

One way to represent that grid in JavaScript is to use an array of Boolean values marking mine locations, where `true` indicates the location of a mine. You could, for example, initialize the variable `mineLocations` to this array by writing the following declaration:

```javascript
let mineLocations = [
    [ true, false, false, false, false, true ],
    [ false, false, false, false, false, true ],
    [ true, true, false, true, false, true ],
    [ true, false, false, false, false, false ],
    [ false, false, true, false, false, false ],
    [ false, false, false, false, false, false ]
];
```

Write a function

```javascript
function countMines(mines)
```

that goes through the array of mines and returns a new array with the same dimensions that indicates how many mines are in the neighborhood of each location. If a location contains a mine, the corresponding entry in the matrix returned by `countMines` should be `-1`. In Minesweeper, the neighborhood consists of the eight adjacent locations as long as those locations are inside the array. For example, the declaration

```javascript
let mineCounts = countMines(mineLocations)
```
should initialize `mineCounts` as follows:

```
-1 1 0 0 2 -1
3 3 2 1 4 -1
-1 -1 2 -1 3 -1
-1 4 3 2 2 1
1 2 -1 1 0 0
0 1 1 1 0 0
```

10. Over the last couple of decades, a logic puzzle called *Sudoku* has become popular throughout the world. In Sudoku, you start with a $9 \times 9$ array of integers in which some of the cells have been filled with a digit between 1 and 9. Your job in the puzzle is to fill each of the empty spaces with a digit between 1 and 9 so that each digit appears exactly once in each row, each column, and each of the smaller $3 \times 3$ squares. Each Sudoku puzzle is carefully constructed so that there is only one solution. For example, Figure 8-10 shows a typical Sudoku puzzle on the left and its unique solution on the right.

Although the algorithmic strategies you need to solve Sudoku puzzles lie beyond the scope of this text, you can easily write a function that checks to see whether a proposed solution follows the Sudoku rules against duplicating values in a row, column, or outlined $3 \times 3$ square. Write a function

```
function checkSudokuSolution(puzzle)
```

that performs this check and returns `true` if the `puzzle` is a valid solution.

![Typical Sudoku puzzle and its solution](image)
11. Write a method `flipHorizontal` that works similarly to the `flipVertical` method presented in the chapter except that it reverses the picture in the horizontal dimension. Thus, if you had a `GImage` containing the image on the left (of Jan Vermeer’s *The Milkmaid*, c. 1659), calling `flipHorizontal` on that image would return a new `GImage` as shown on the right:

![Image before flip](image1.png)
![Image after flip](image2.png)

12. Write a method `rotateLeft` that takes a `GImage` and produces a new `GImage` in which the original has been rotated 90 degrees to the left.
[I remember the exact moment] when the concept of “inheritance” (or classes and subclasses) had been created. I realized immediately that this was the solution to a very important problem Ole-Johan Dahl and I had been struggling with for months and weeks. And sure enough, inheritance has become a key concept in object-oriented programming, and thus in programming in general.

—Kristen Nygaard, address at the IRIS 19 conference, 1996

Norwegian computer scientists Kristen Nygaard and Ole-Johan Dahl developed the central ideas of object-oriented programming more than 50 years ago as part of their work on the programming language SIMULA. Early versions of SIMULA appeared in the early 1960s, but the stable version of the language that brought these concepts to the attention of the world appeared in 1967. The initial work on SIMULA was carried out at the Norwegian Computing Center, a state-funded research laboratory in Norway focusing on developing better software-engineering techniques. Both later joined the faculty at the University of Oslo. Although their work took several decades to become established in the industry, interest in object-oriented techniques has grown considerably in the last three decades, particularly after the release of modern object-oriented languages like C++ and Java. For their contributions, Nygaard and Dahl received both the 2001 Turing Award from the Association for Computing Machinery and the 2001 John von Neumann Medal from the Institute of Electrical and Electronic Engineers.
Objects

When you learned about arrays in Chapter 8, you took your first steps toward understanding an extremely important idea in computer programming: the use of compound data structures to represent complex collections of information. When you declare an array in the context of a program, you are able to combine an arbitrarily large number of data values into a single structure that has conceptual integrity as a whole. If you need to do so, you can select particular elements of that array and manipulate them individually. But you can also treat the array as a unit and manipulate it all at once.

The ability to take individual values and organize them into coherent units is one of the fundamental features of modern programming languages. Functions allow you to unify many independent operations under a single name. Compound data structures—of which arrays are only one example—offer the same facility in the data domain. In each case, being able to aggregate the tiny pieces of a program into a single, higher-level structure provides both conceptual simplification and a significant increase in your power to express ideas in programming. The power of unification is hardly a recent discovery; it has given rise to social movements and to nations, as reflected in the labor anthem that proclaims “the union makes us strong” and the motto “E Pluribus Unum”—“out of many, one”—on the Great Seal of the United States.

Although arrays are a powerful tool when you need to model real-world data that can be represented as a list of ordered elements, it is also important to be able to combine unordered data values into a single unit. This chapter describes how JavaScript supports such assembled collections of data values and how to use those values effectively in programs.

9.1 Objects in JavaScript

One of the challenges in explaining JavaScript’s approach to structured data is the fact that JavaScript uses the term object to refer to more than one idea. In languages like Java and C++ that support the object-oriented paradigm developed by Kristen Nygaard and Ole-Johan Dahl, the word object refers to a structure that encapsulates data and behavior. As you will see later in this chapter, JavaScript allows objects to assume that role. JavaScript, however, also uses the word object to refer to an older data model that is traditionally called a structure or an aggregate. This chapter begins by exploring this more primitive model and then builds the more modern concept of an object on top of that foundation, just as JavaScript itself does.

Objects as data aggregates

Objects in JavaScript are similar to arrays in that they allow multiple data values to be considered as a unit. The fundamental difference lies in how the individual data
values are identified. In an array, each of the elements is identified by an index number. In an object, each of the internal components, which are generally called fields, is identified by a name.

Objects are useful whenever you have to model a collection of data that has individual components but nonetheless represents an integrated whole. If, for example, you are designing a payroll system for a company, each individual employee has a variety of attributes such as name, job title, and salary, but it still makes sense to think of all those components as a single entity that describes a particular employee. At the rather small firm of Scrooge and Marley that appears in *A Christmas Carol* by Charles Dickens, the data for the two employees might look something like this:

```
<table>
<thead>
<tr>
<th>name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ebenezer Scrooge</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>title</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CEO</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>salary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>£1000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob Cratchit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>title</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>clerk</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>salary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>£25</td>
<td></td>
</tr>
</tbody>
</table>
```

Each of the two objects in this diagram is an aggregate of three distinct parts: a *name* field indicating the name of the employee, a *title* field indicating the job title, and a *salary* field indicating the annual compensation. Programs can treat these employee objects at either of two levels. At the holistic level, an employee acts as a single data value. You can assign it to a variable, pass it as a parameter, or return it as a result. When you need to take a more reductionistic view, you can select and manipulate the individual fields.

**Creating objects**

JavaScript makes it much easier to create compound objects than most other modern languages. All you need to do is enclose a set of field specifications inside curly braces. Each field specification consists of the field name and its value separated by a colon, with the field specifications themselves separated by commas. The following line, for example, declares the variable `clerk` containing the information for Bob Cratchit:

```javascript
let clerk =
{ name: "Bob Cratchit", title: "clerk", salary: 25 };
```

This syntactic format for creating objects is called *JavaScript Object Notation*, which is typically shortened to *JSON*. 
Selecting fields

Given a JavaScript object, you can select individual fields using the *dot operator*, which is written in the form

\[ \text{object.name} \]

where *object* specifies the object as a whole and *name* specifies the desired field. Thus, given the declaration of *clerk* in the preceding section, you could select the *name* field by writing

\[ \text{clerk.name} \]

which in this case has the string value "Bob Cratchit". Fields are also assignable. For example, when the reformed Mr. Scrooge tells Bob Cratchit, “I am about to raise your salary,” he could do so by writing

\[ \text{clerk.salary += 5;} \]

which gives the underpaid clerk an extra five pounds a year.

Objects as references

Like arrays, objects in JavaScript are treated as references to the actual value. As a result, any changes made to the fields of an object passed as a parameter to a function will still be evident after that function returns. For example, if Ebenezer Scrooge decided to be even more generous, he could define the following function, which would double the salary of an employee:

\[
\text{function doubleSalary(employee) {}
    \text{employee.salary *= 2;}
}\]

Calling

\[ \text{doubleSalary(clerk);} \]

would have the desired effect of doubling the salary field in the object contained in the variable *clerk*.

9.2 Defining a type to represent points

As noted in the preceding section, one of the advantages of using objects is that doing so makes it possible to combine several related pieces of information into a composite value that can be manipulated as a unit. An important practical application of this principle arises when you need to represent a point in a two-dimensional space, which might be the drawing surface of a graphics window.
So far, the graphical programs in this text have kept track of independent x and y coordinates, which is sufficient for many applications. When you start to work with more complex graphical programs, however, it is useful to store the x and y values in an integrated unit called a point.

Combining the x and y coordinates into a single object makes it possible to work with points as composite values, which means that you can work with them just as you work with any other data value. You can assign a point to a variable, create an array of points, pass a point as an argument to a function, and return a point as a result. This last example—returning a point as the result of a function call—adds a new capability that would otherwise be difficult to achieve. A JavaScript function is allowed to return only a single value, so there is no way for a function to return the x and y coordinates independently. A function, however, can return a point, which acts as a single value. The caller can then extract its x and y coordinates.

**Strategies for creating points**

The expressiveness of the JSON model for creating objects makes it possible to use points in an application without defining a new type. For example, the following declarations define two point-valued variables:

```javascript
let origin = { x: 0, y: 0 };  
let lowerRight = { x: GWINDOW_WIDTH, y: GWINDOW_HEIGHT };  
```

The first declaration defines the variable `origin` to be the point (0, 0) at the upper left corner of the window; the second defines the variable `lowerRight` to be the point in the lower right corner, assuming that the constants `GWINDOW_WIDTH` and `GWINDOW_HEIGHT` are set up as they have been in the earlier graphical examples. Once you have these variables, you can create a `GLine` that runs diagonally across the window like this:

```javascript
GLine(origin.x, origin.y, lowerRight.x, lowerRight.y)  
```

Although many JavaScript applications rely entirely on the JSON model, it often improves the structure of your code to provide a function that creates the new object. Doing so has a couple of advantages. Most importantly, the name of the function that creates the new object also serves as a name for the type of value that function creates. For example, the factory method

```javascript
function Point(x, y) {  
    return { x: x, y: y };  
}
```

creates and returns an object whose conceptual type is `Point`, even though there is no difference internally between objects created using the `Point` function and those
Objects

300

created using the JSON model. In keeping with the convention for defining factory methods that serve as types, the name `Point` begins with an uppercase letter, just as `GRect` and `GOval` do.

It might at first seem confusing to see field definitions like `x: x` in the `return` statement. The `x` that comes before the colon specifies the name of the field; the `x` that comes after the colon is an expression indicating the value, which in this case is the argument `x`. Factory methods will often contain expressions in which the field name and the field value match.

Given this definition of `Point`, the declarations for the variables `origin` and `lowerRight` can be rewritten like this:

```javascript
let origin = Point(0, 0);
let lowerRight = Point(GWINDOW_WIDTH, GWINDOW_HEIGHT);
```

Factory methods can perform any computations necessary to calculate the values stored in the fields of the new object. For example, `Point` can be rewritten so that calling `Point` with no arguments automatically creates a point at the origin:

```javascript
function Point(x, y) {
  if (x === undefined) {
    x = 0;
    y = 0;
  }
  return { x: x, y: y };}
```

**Using points in an application**

Point-valued objects are often useful in graphical programs, particularly if you need to store coordinate pairs in an array. For example, the `YarnPattern.js` program in Figure 9-1 creates beautiful graphical patterns using only `GLine` objects. Each of the `GLine` objects connects two points stored in an array using a process that models one that you can easily carry out in the real world. You begin by taking a rectangular board and arranging pegs around the perimeter so that they are evenly spaced along all four edges, as in the following small example in which the pegs are numbered clockwise from the upper left corner:
9.2 Defining a type to represent points

![Program to simulate threading yarn around a series of pegs](image)

```javascript
/*
 * File: YarnPattern.js
 * -------------------
 * This program uses the GLine class to simulate the process of winding a
 * piece of colored yarn around a set of pegs equally spaced along the
 * perimeter of a rectangle. At each step, the yarn is stretched from
 * its current peg to the one DELTA pegs further on.
 */

/* Constants */

const PEG_SEP = 12;
const N_ACROSS = 80;
const N_DOWN = 50;
const GWINDOW_WIDTH = N_ACROSS * PEG_SEP;
const GWINDOW_HEIGHT = N_DOWN * PEG_SEP;
const DELTA = 113;

/*
 * Creates a pattern that simulates the process of winding a piece of
 * yarn around an array of pegs along the edges of the graphics window.
 */

function YarnPattern() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let pegs = createPegArray();
    let thisPeg = 0;
    let nextPeg = -1;
    while (thisPeg !== 0 || nextPeg === -1) {
        nextPeg = (thisPeg + DELTA) % pegs.length;
        let p0 = pegs[thisPeg];
        let pl = pegs[nextPeg];
        let line = GLine(p0.x, p0.y, pl.x, pl.y);
        line.setColor("Magenta");
        gw.add(line);
        thisPeg = nextPeg;
    }
}

/*
 * Creates a new Point object. If this function is called with no
 * arguments, it creates a Point object at the origin.
 */

function Point(x, y) {
    if (x === undefined) {
        x = 0;
        y = 0;
    }
    return { x: x, y: y };```
From here, you create a figure by winding a single piece of yarn through the pegs, starting at peg 0 and then moving ahead `DELTA` spaces on each cycle. For example, if `DELTA` is 11, the yarn goes from peg 0 to peg 11, then from peg 11 to peg 22, and then (counting past the beginning) from peg 22 to peg 5, as follows:

The process continues until the yarn returns to peg 0, creating the following pattern:
The program in Figure 9-1 begins by calling `createPegArray` to create the array of points around the perimeter. The code creates pegs from left to right across the top, from top to bottom along the right side, from right to left across the bottom, and finally from bottom to top along the left side. When `createPegArray` returns, the `YarnPattern` program starts at peg 0 and then advances `DELTA` steps each cycle until the index loops back to 0. On each cycle, `YarnPattern` creates a `GLine` object to connect the current point in the array with the previous one. Figure 9-2 shows a larger image using the parameters shown in the program listing.

### 9.3 Libraries and interfaces

You have been working with objects in this book ever since Chapter 2 when you wrote your first programs using the graphics library. The `GObject` subclasses all bear at least some resemblance to the `Point` type introduced in section 9.2, although there are also some important differences. Just like the `Point` factory method used to create a new object whose conceptual type is `Point`, the factory methods `GRect`, `GOval`, and `GLine` create graphical objects of the appropriate type. The most obvious difference lies in the way clients refer to component values within the object. Given a `Point` object as it is currently defined, clients select the
x and y fields using the dot operator. Given a GRect object, clients obtain its coordinates by calling the methods `getX` and `getY`.

Modern programming practice strongly favors the method-based model over using the dot operator to select internal fields explicitly. Using methods in some sense mediates the interaction between the client and the implementation. If a client has direct access to the components of an object, there is always some danger that a careless or malicious client will use that access to violate some condition on which the implementation depends. Most object-oriented languages make it easy for an implementation to prevent clients from gaining direct access to the internal fields, thereby providing a level of security through the language design.

Even though blocking direct access to the internal fields of an object is not really possible in JavaScript, it nonetheless makes sense to define methods that offer a model of how the implementation intends clients to make use of that data. Well-behaved clients will use those methods and not try to select the underlying fields directly.

As it happens, a GRect object does have fields called x and y that contain the coordinates of the rectangle. If, for example, you declared a GRect object as

```
let rect = GRect(150, 50, 200, 100);
```

you could—if you were being ornery—select the x coordinate 150 by writing

```
rect.x
```

As the bug symbol implies, doing so is a bad idea. Although JavaScript may allow it, selecting the internal components of an object violates the implied contract between the client and the implementer. The implementer has provided a method called `getX` for this purpose, and it is inappropriate for clients to look inside the implementation and discover just where this value is being stored. It should be perfectly legitimate for the implementer to change the name of that internal variable as long as the `getX` method continues to return the correct value. If clients cheat and rely on details that are specific to a particular implementation, they have no right to complain if changes to the implementation end up breaking their programs.

In computer science, the implied contract between a client and an implementer is called an interface. Conceptually, an interface contains the information that clients need to know about a library—and no more. For clients, getting too much information can be as bad as getting too little, because additional detail is likely to make the interface more difficult to understand. Often, the real value of an interface lies not in the information it reveals but rather in the information it hides.
When you design an interface for a library, you should try to protect the client from as many of the complicating details of the implementation as possible. In doing so, it is perhaps best to think of an interface not as a communication channel between the client and the implementation, but instead as a wall that divides them.

Like the wall that divided the lovers Pyramus and Thisbe in Greek mythology, the wall representing an interface contains a chink that allows the client and the implementation to communicate. In programming, that chink exposes the method definitions that allow the client and implementation to communicate. The main purpose of the wall, however, is to keep the two sides apart. Ideally, all the complexity involved in the realization of a library lies on the implementation side of the wall. The interface is successful if it keeps that complexity away from the client side. Interfaces therefore provide a tool to support the principle of information hiding introduced in Chapter 5.

Unlike programming languages that emphasize the principle of information hiding, JavaScript does not offer any formal mechanism for defining an interface. In JavaScript, an interface is purely an idea. That idea is realized only in the design of a library and in its documentation. When you design a library, however, it is important to understand what you want its interface to make available to the client and what details you want that interface to conceal.

### 9.4 Adding methods to objects

If you want to follow the principles of modern programming style, you need to learn how to add methods to a JavaScript object. For the `Point` type defined in section 9.2, you want to add the methods `getX` and `getY` so that clients can obtain the coordinates of the point without referring directly to the `x` and `y` fields. Thus, instead of having clients write

```javascript
pt.x
```

you want to make it possible for them to write

```javascript
pt.getX()
```

Methods that return the value of a field are called `getters`. Methods that change the value of a field, which are far less common, are called `setters`. 
Adding methods to the `Point` type—and in the process changing it into a `Point` class—is not nearly as hard as you might at first imagine. The receiver syntax used for method calls looks very much like the operation of selecting a field from an object and is in fact implemented in precisely that way. The name `getX` is simply a field in the `Point` class, just as `x` is. The difference is that the value of the `getX` field is a function that returns the value of the internal `x` component. JavaScript functions are, after all, first-class objects, and it is perfectly appropriate to store a function in the field of an object. The only complication in writing the definition of the `getX` function is that its implementation must have some way of gaining access to the value of the `x` component of the point.

One very common—but by no means the best—approach in JavaScript is to store the `x` and `y` as part of the object and then to use the special keyword `this` as a reference to the current object. In JavaScript, any method call that uses the receiver syntax `receiver.name(arguments)` stores a reference to `receiver` in the keyword `this` so that the receiver is available inside the implementation of the method. For example, if you call

```javascript
pt.getX()
```

the implementation of `getX` can refer to the `x` field of the receiver as `this.x`. This strategy allows you to define the following implementation of the factory method `Point`, which exports a structure containing fields named `x` and `y` along with methods for obtaining the values of those fields using the receiver syntax:

```javascript
function Point(x, y) {
  if (x === undefined) { x = 0; y = 0; }
  return {
    x: x,
    y: y,
    getX: function() { return this.x; },
    getY: function() { return this.y; }
  };
}
```

Although this implementation is technically correct, it represents such a violation of the principles of object-oriented design as to merit the bug symbol. The problem is that this version of `Point` leaves the fields `x` and `y` exposed to the client. Given a point `pt`, the client need not call `pt.getX()` to obtain the `x` coordinate, but can instead simply write `pt.x`, bypassing the `getX` method altogether. Worse still, the client can reach in and change the values of the `x` and `y` fields without going through methods in the class. Allowing this kind of access completely undermines the principle of information hiding.
The key to a better strategy lies in recognizing that any function defined inside the `Point` function already has access to the values of `x` and `y` because they are part of the closure. Thus, `getX` and `getY` can simply return these values without using the keyword `this` at all, as illustrated in the following version of `Point`:

```javascript
function Point(x, y) {
    if (x === undefined) { x = 0; y = 0; }
    return {
        getX: function() { return x; },
        getY: function() { return y; }
    };
}
```

All classes in this book use this closure-based approach, which is adapted from a model recommended by Doug Crockford, who was introduced on page 39. This strategy has the following advantages:

1. It completely hides the underlying representation from the client.
2. It eliminates the need for the `this` keyword, which is often confusing to use.
3. It results in shorter factory methods that are easier to read.

**The `toString` method**

The one remaining feature that is useful to add to the definition of the `Point` class is a method that defines how the object should be converted to a string. Points are conventionally enclosed in parentheses with the two coordinate values separated by a comma, as in the point `(0, 0)` that represents the origin. If you want to display the value of a `Point`, it would be ideal if you could have JavaScript use this same form.

When JavaScript needs to determine how an object should be represented as a string, it checks to see whether that object includes a method called `toString`. If it does, JavaScript calls that method and then uses the result as the string value. Adding `toString` to the `Point` class gives rise to the final version of the `PointClass.js` file, which appears in Figure 9-3.

Applications that want to use the `Point` class can do so simply by importing `PointClass.js` as a source library, as follows:

```javascript
import "PointClass.js";
```

You could, for example, add this line to the implementation of `YarnPattern` in Figure 9-1, but you should also change the program so that it uses the `getX` and `getY` methods instead of making direct references to the `x` and `y` fields.
Implementing inheritance

As you know from the example of the graphics library introduced in Chapter 2, classes in an object-oriented language form hierarchies in which subclasses inherit behavior from the superclasses. JavaScript allows you to achieve the same effect by having the factory method for the subclass call the factory method for the superclass to obtain an object, add in the new behavior appropriate to the subclass, and then return the object with the modified behavior.

Defining a class hierarchy for employees

Although the simple model for keeping track of employee data used in section 9.1 might work for a two-person firm like Scrooge and Marley, large companies have different classes of employees that are similar in some ways but different in others. For example, a company might have hourly, commissioned, and salaried employees. Those employee categories will share some information, so that it makes sense to define methods like getName and getJobTitle that work for all employees. By contrast, calculating the pay for each class of employee differs according to the employee class. A getPay method must therefore be implemented separately for each employee type. This model suggests that the class hierarchy used to represent employees might look something like the UML diagram in Figure 9-4.

The root of this hierarchy is the Employee class, which defines the methods that are common to all employees. The Employee class therefore exports methods like

```javascript
function Point(x, y) {
    if (x === undefined) { x = 0; y = 0; }
    return {
        getX: function() { return x; },
        getY: function() { return y; },
        toString: function() { return "(" + x + ", " + y + ")"; }
    }
}
```
getName and getJobTitle, which the other classes simply inherit. Conversely, it is almost certainly necessary to write separate getPay methods for each of the subclasses, because the computation is different in each case. The pay of an hourly employee depends on the hourly rate and the number of hours worked. For a commissioned employee, the pay is typically the sum of some base salary plus a commission on the sales volume for which that employee is responsible. At the same time, it is useful to note the fact that every employee has a getPay method, even though its implementation differs for each of the subclasses. The UML diagram therefore includes a getPay method at the level of the Employee class, even though that method is defined at a lower level. The names of the Employee class and the getPay method within it are set in italic type to indicate that these are abstract entities that act as placeholders for the concrete definitions in the subclasses.

Figure 9-5 defines a simple version of the Employee class definition. The factory method takes two parameters, name and title, and creates an object that contains these fields along with the methods that are common across all subclasses. The object includes an entry for getPay, which has the following value:

```javascript
function() {
    alert("getPay not defined for Employee class");
}
```

This built-in function alert brings up a dialog box containing the string value passed as an argument. In this case, the alert box lets the client know that the getPay method has been called on the Employee class instead of on one of its subclasses, which is a violation of the interface for the class hierarchy. Each of the
Objects subclasses will replace this version of the `getPay` method with one that calculates the employee’s pay appropriately. In object-oriented programming, the process of providing a new definition for an existing method is called **overriding**.

The code for the `HourlyEmployee` class appears in Figure 9-6. The factory method begins by calling `Employee` to set up the common parts of the structure and then declares the local variables it requires. It then overrides the `getPay` method with the following version, which is appropriate for employees who are paid hourly (you will have a chance to code the other subclasses in exercise 2):

```javascript
function() { return hoursWorked * hourlyRate; }
```

**Creating a graphical class hierarchy**

In the discussion of decomposition in section 5.4, the primary example was the problem of drawing a train consisting of different types of cars, like this:
Although the `DrawTrain.js` program was useful as an exercise in decomposition, the result isn’t helpful if you want to add animation to the program. To make the train appear to move, you would need to have your program change the position of every graphical object in the diagram on each time step. It would be better if the train were a `GCompound` you could animate as a single unit.

The strategy of decomposition is not limited to functions. In many cases, it is equally useful to decompose the problem by creating a hierarchy of classes whose structure reflects the relationship among the objects. In this problem, it makes sense to define a `Train` class as a subclass of `GCompound` so that it acts as a single graphical object. The individual components inside a train can then be objects of a class called `TrainCar`, which is also a subclass of `GCompound`. The three different types of train cars are then subclasses of `TrainCar`. It would also be convenient for clients if the `Train` class included a method that allows you to add a `TrainCar` to the end of the train. That method can’t be called `add` because `GCompound` has an `add` method this application needs, so this example uses `append` for that purpose.

This strategy for representing the train gives rise to the UML hierarchy shown in Figure 9-7. Every class in the hierarchy is a `GCompound` and can therefore be displayed on the graphics window. The code for the new version of `DrawTrain`, which includes code to move the train after a mouse click, appears in Figure 9-8.
/*
 * File: DrawTrain.js
 * ___________
 * This program defines a class hierarchy for representing train cars
 * based on the Compound class. The code in this file implements only
 * the Boxcar class; implementation of the Engine and Caboose classes
 * are left to the reader as exercises.
 */

/* Constants */
const GWINDOW_WIDTH = 500; /* Width of the graphics window */
const GWINDOW_HEIGHT = 300; /* Height of the graphics window */
const CAR_WIDTH = 113; /* Width of the frame of a train car */
const CAR_HEIGHT = 54; /* Height of the frame of a train car */
const CAR_BASELINE = 15; /* Distance of car base to the track */
const CONNECTOR = 6; /* Width of the connector on each car */
const WHEEL_RADIUS = 12; /* Radius of the wheels on each car */
const WHEEL_INSET = 24; /* Distance from frame to wheel center */
const CAB_WIDTH = 53; /* Width of the cab on the engine */
const CAB_HEIGHT = 12; /* Height of the cab on the engine */
const SMOKESTACK_WIDTH = 12; /* Width of the smokestack */
const SMOKESTACK_HEIGHT = 12; /* Height of the smokestack */
const SMOKESTACK_INSET = 12; /* Distance from smokestack to front */
const DOOR_WIDTH = 27; /* Width of the door on the boxcar */
const DOOR_HEIGHT = 48; /* Height of the door on the boxcar */
const CUPOLA_WIDTH = 53; /* Width of the cupola on the caboose */
const CUPOLA_HEIGHT = 12; /* Height of the cupola on the caboose */
const TIME_STEP = 20; /* Time step for the animation */

/* Main program */

function DrawTrain() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let train = Train();
    train.append(Boxcar("Green"));
    let x = (gw.getWidth() - train.getWidth()) / 2;
    let y = gw.getHeight();
    gw.add(train, x, y);
    let timer = null;

    function clickAction() {
        timer = setInterval(step, TIME_STEP);
    }

    function step() {
        train.move(-1, 0);
        if (train.getX() + train.getWidth() < 0) {
            clearInterval(timer);
        }
    }
    gw.addEventListener("click", clickAction);
}

/*
function Train() {
  let train = GCompound();
  train.append = function(car) {
    train.add(car, train.getWidth(), 0);
  };
  return train;
}

function TrainCar(color) {
  let frame = GCompound();
  let x = CONNECTOR;
  let y = -CAR_BASELINE;
  frame.add(GLine(0, y, CAR_WIDTH + 2 * CONNECTOR, y));
  frame.add(Wheel(), x + WHEEL_INSET, -WHEEL_RADIUS);
  frame.add(Wheel(), x + CAR_WIDTH - WHEEL_INSET, -WHEEL_RADIUS);
  let r = GRect(x, y - CAR_HEIGHT, CAR_WIDTH, CAR_HEIGHT);
  r.setFilled(true);
  r.setFillColor(color);
  frame.add(r);
  return frame;
}

function Boxcar(color) {
  let boxcar = TrainCar(color);
  let x = CONNECTOR + CAR_WIDTH / 2;
  let y = -(CAR_BASELINE + DOOR_HEIGHT);
  boxcar.add(GRect(x - DOOR_WIDTH, y, DOOR_WIDTH, DOOR_HEIGHT));
  boxcar.add(GRect(x, y, DOOR_WIDTH, DOOR_HEIGHT));
  return boxcar;
}

function Wheel() {
  let wheel = GCompound();
  let r = WHEEL_RADIUS;
  let circle = GOval(-r, -r, 2 * r, 2 * r);
  circle.setFilled(true);
  circle.setFillColor("Gray");
  wheel.add(circle);
  return wheel;
Given this design, you can assemble the three-car train shown in the example by writing the following code:

```javascript
let train = Train();
train.append(Engine());
train.append(Boxcar("Green"));
train.append(Caboose());
```

The first line creates an empty train, and the remaining lines add an engine, a green boxcar, and a caboose to the end of the train.

### 9.6 Using objects as maps

So far, you have selected the fields in an object using the dot operator, which requires you to know the name of the field at the time you write the program. For certain applications, it is useful to select a field whose name is not known until the program runs. JavaScript allows that form of selection as well, which has the form

```javascript
object["name"]
```

where `object` is the object from which the selection is made and `name` is a string expression indicating the name of the field. For example, if `pt` is an instance of the `Point` class introduced earlier in the chapter, you can select its `x` component either by writing `pt.x` in the now-familiar way or by writing `pt["x"]`. Both forms select the field named "x" from the object stored in `pt`.

The bracketed form of object selection is typically used to implement a data structure that computer scientists call a **map**, which is conceptually similar to a dictionary. A dictionary allows you to look up a word to find its meaning. A map is a generalization of this idea that provides an association between an identifying tag called a **key** and an associated **value**, which is often a much larger and more complicated structure. In the dictionary example, the key is the word you’re looking up, and the value is its definition.

Maps have many applications in programming. For example, an interpreter for a programming language needs to be able to assign values to variables, which can then be referenced by name. A map makes it easy to maintain the association between the name of a variable and its corresponding value.

### Using maps in an application

If you fly at all frequently, you quickly learn that every airport in the world has a three-letter code assigned by the International Air Transport Association (IATA). For example, the John F. Kennedy airport in New York City is assigned the three-letter code JFK. Other codes, however, are considerably harder to recognize.
Most web-based travel systems offer some means of looking up these codes as a service to their customers.

A simple way to implement this facility is to create a map whose keys are the airport codes and whose values are the city names. If this map is stored in the constant \texttt{AIRPORT\_CODES}, all you need to do to find the city corresponding to the three-letter airport code is evaluate the expression

\[
\text{AIRPORT\_CODES[code]}
\]

If \texttt{AIRPORT\_CODES} contains a field matching the three-letter code for the airport, the expression will return the corresponding city name from the map. If there is no field matching the code, the value of the expression is \texttt{undefined}.

The interesting question is how to initialize \texttt{AIRPORT\_CODES}. In JavaScript, the simplest approach is to initialize \texttt{AIRPORT\_CODES} explicitly as part of the program by providing its JSON representation. The \texttt{AirportCodes.js} program in Figure 9-9 includes the following definition of \texttt{AIRPORT\_CODES}, in which the airports appear in decreasing order by passenger volume:

```javascript
const AIRPORT\_CODES = {
    ATL: "Atlanta, GA, USA",
    ORD: "Chicago, IL, USA",
    LHR: "London, England, United Kingdom",
    HND: "Tokyo, Japan",
    LAX: "Los Angeles, CA, USA",
    CDG: "Paris, France",
    DFW: "Dallas/Ft Worth, TX, USA",
    FRA: "Frankfurt, Germany",
    PEK: "Beijing, China",
    MAD: "Madrid, Spain",
    . . . over 2500 more airport codes . . .
}
```

The complete list of three-letter codes is long but still fits easily in a JavaScript file.

The main program in Figure 9-9 reads three-letter codes from the user and displays the corresponding city name, as shown in the following sample run:

<table>
<thead>
<tr>
<th>Airport code: LHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>London, England, United Kingdom</td>
</tr>
<tr>
<td>Airport code: LAX</td>
</tr>
<tr>
<td>Los Angeles, CA, USA</td>
</tr>
<tr>
<td>Airport code: XXX</td>
</tr>
<tr>
<td>There is no airport code XXX</td>
</tr>
<tr>
<td>Airport code:</td>
</tr>
</tbody>
</table>
Computers got their start as machines designed to process data. Today, programs often use data not as passive information to be processed but instead to control the program’s entire operation. Programs that allow data to control their execution are said to be data driven. Data-driven programs are usually shorter, more flexible, and easier to maintain than programs that incorporate the same information directly into the program design.
Programmed instruction courses

Forty years or so ago, when computers were just starting to be used as general-purpose tools, there was a movement within the educational community to use computers as part of the teaching process—a movement that has seen new life with the Massive Open Online Courses (MOOCs) that caused quite a stir several years ago. Historically, one of the leading strategies for computer-aided education was programmed instruction, a process in which a computerized teaching tool asks a series of questions so that previous answers determine the order of subsequent questions. As long as a student is getting all the right answers, the programmed instruction process skips the easy questions and moves quickly on to more challenging topics. For the student who is having trouble, the process moves more slowly, leaving time for repetition and review.

Suppose you have been assigned the problem of writing an application that makes it possible to present material in a programmed instruction style. In a nutshell, your program must be able to

• Ask the student a question
• Get an answer from the student
• Move on to the next question, which depends on the student’s response

Such a program will inevitably be simpler than a commercial application, but you can easily design a program that illustrates the general principles involved.

The importance of using a flexible data structure

It is possible to design a programmed-instruction application as a set of functions. Each function asks a question, reads in an answer, and then calls another function appropriate to the answer the student supplies. Such a program, however, would be difficult to change. Someone who wanted to add questions or design an entirely new course would need to write new functions. Writing functions is simple enough for someone who understands programming, but not everyone does. Most programmed-instruction courses are designed by teachers in a specific discipline who are usually not programmers. Forcing them to work in the programming domain limits their ability to use the application.

As the programmer on the project, your goal is to develop an application that presents a programmed instruction course to the student but allows teachers without programming skills to supply the questions, expected answers, and cross-reference information so that your application can present the questions in the appropriate order. To do so, the best approach is to design your application as a general tool that takes all data pertaining to the programmed instruction course from a file. If
you adopt this approach, the same program can present many different courses by using different data files.

In designing the application, you need to start by considering such broad questions as:

- **What are the overall requirements of the general problem?** In particular, you need to understand the set of operations your program must support, apart from any specific domain of instruction.

- **How can you represent the data for the programmed instruction course in the context of your program?** As part of the design phase, you need to develop a data structure consisting of new class definitions that implement conceptual types such as arrays and maps.

- **What should a course data file look like?** As you make this decision, you need to keep in mind that the data file is being edited by nonprogrammers whose expertise is in the specific domain under consideration.

- **How do you convert the external representation used in the data file into the internal one?**

- **How do you write the program that manipulates the internal data structures?**

The rest of this chapter considers each of these questions in turn.

**Framing the problem**

At one level, it is easy to outline the operation of the program. When your program runs, its basic operation is to execute the following steps repeatedly:

1. Ask the student the current question. A question consists of one or more lines of text, which you can represent as strings.
2. Request an answer from the student, which can also be represented as a string.
3. Look up the answer in a table of possibilities provided for that question. If the answer appears in the table, consult the data structure to choose what question should become the new current question. If the student’s answer does not match any of the possibilities provided by the course data file, the student should be informed of that fact and given another chance at the same question.

Many details are missing from this outline, but it is a start. Even at this level, the outline provides some insight into the eventual implementation. For example, you know that you need to keep track of what the “current question” is. To do so, it makes sense to number the questions and then store the current question number in a variable.
Writing the program turns out to be one of the easier pieces of the task; the harder problems arise in representing the data structure. For the program to be general and flexible, all the information that pertains to an actual course must be stored in a data file, not built directly into the program. The program’s job is to read that data file, store the information in an internal data structure, and then process that structure as outlined earlier in this section. Thus, your next major task is to design the data structures required for the problem so that you have a context for building the program as a whole.

The process of designing the data structures has two distinct components. First, you have to design an internal data structure for use by the program. The internal data structure consists of class definitions that use JavaScript’s data structures so that the resulting classes mirror the organization of the real-world information you seek to represent. Second, you must design an external data structure that indicates how the information is stored in the data file. These two processes are closely related, mostly because they each represent the same information. Even so, the two structures are tailored to meet different purposes. The internal structure must be easy for the programmer to use. The external structure must make it easy for someone to write a course, without making it too difficult for the programmer.

**Designing the internal representation**

The first step in the process is to design a data structure that incorporates the necessary information. In most cases, it is easiest to design the data structure from the top down, starting with the highest-level structure and then refining it by specifying more and more of the details. Here, the highest level is the course as a whole. That course contains a set of questions, and each question contains a set of answers and their associated transitions.

In designing any data structure, one of the most important concepts is that of *encapsulation*, which is the process of combining related pieces of data into structures that can be treated as complete units. For most applications, the encapsulation process is hierarchical and must be considered at varying levels of detail. At the highest level, you think of the entire structure as a single object, which contains all the information you will need for the course. That object is a reference to some data collection, the details of which are not important at the highest level of detail:
Whenever you need to pass the entire course to a function, all you have to do is pass the variable `course`, which is a small, easily manipulated reference that gives you access to the other data. It is only when a function needs to manipulate the individual fields of the data structure that it has to look inside the structure to see its details.

Given that your current purpose is to design the internal structure, however, you do need to understand what is contained within the dotted box in this diagram. Intuitively, you know that the data structure as a whole contains a list of questions, although it may make sense to include other information as well, such as the title of the course. At first glance—although it will make sense to reconsider this design later—it seems as if the questions should be an array of some as-yet-undefined class representing a single question. Thus, the data structure at the current level of decomposition looks like this:

![Diagram of course structure]

Each question, however, must store the text of the question along with some structure that associates possible responses with next-question indicators. Thus, the entire structure will look something like the diagram in Figure 9-10.

![Figure 9-10: Rough design of the data structure for the teaching machine]
Designing the external structure

Before you turn to the details that will allow you to code the internal data structures, it often helps to think about how to represent the information within the data file. Files are simply text, and the organization provided by the data structure hierarchy in JavaScript must be expressed in the design of the file format. The file structure design must also make it easy for someone to write and edit, even if that person is not a programmer. Thus, you should choose a representation that is as simple as possible. In this case, it seems easiest to write out each question, one after another, along with its likely answers. So that the computer can tell where the one question stops and the next question begins, you must define some convention for separating the question-and-answer units. A blank line works well in this context, as it does in most file structures. Thus, in individual units separated by blank lines, you have the data for each question and its answers.

But what goes into each question-and-answer unit? First of all, you need the text of the question, which consists of individual lines from the file. You also need some way to indicate the end of the question text, and the easiest way, both for you and for the course designer, is to define some kind of marker for the end, such as a line of five dashes. Furthermore, you must allow the course designer to specify the mapping between each answer and the number of the next question. One possibility is to represent both of these values on a single data line consisting of the answer text, followed by a colon, followed by the index of the next question. Other formats are possible, but this design seems as if it would be easy for a novice to learn. Thus, the data for an individual question entry in the file looks like this:

True or false: The earth revolves around the sun.
-----
true: 3
table: 2

The question text consists of a single line, after which there are two acceptable answers. If the user types in false, the program should go to question 2; if the user types in true, it should move on to question 3.

This example brings up an interesting question. How do you assign question numbers to each of these entries? One approach would be to arrange the questions in the file and number them sequentially. While this strategy might make life easy for you as the programmer, it makes life difficult for the person writing a course.

To understand why this is so, suppose that the course designer wants to add a new question near the beginning of an existing course. All subsequent questions move down by one, all the question numbers change, and the course designer has to spend a considerable amount of time renumbering all the next-question indicators.
A better approach is to let the person who writes the question give it a number. For example, if the sample question about the earth and sun were question #1 in the course file, its entry in the file would begin with its question number, as follows:

```
1
True or false: The earth revolves around the sun.
-----
ture: 3
false: 2
```

The advantage of allowing the course designer to supply question numbers is that it makes editing the course much easier. Someone who wants to add a new question can just give it a question number that hasn’t been used before. None of the other question numbers need to change. The course designer can then insert the new question anywhere in the data file, because there is no longer any reason that the question numbers need to be consecutive.

One implication of allowing the course designer to choose question numbers is that those numbers need not be consecutive. In fact, it generally makes designing a course easier if the numbers are not consecutive, because then you can assign numbers so that they reflect the logical structure of the course. For example, you could decide that questions numbered between 1 and 100 were part of one topic, questions between 101 and 200 were part of another topic, and so on. In all likelihood, you wouldn’t fill up these ranges, so there would be gaps in the sequence.

The idea that the question numbers are not necessarily consecutive has implications for the data structure as well. If the question numbers are completely arbitrary and don’t indicate the sequence of the question, then the structure that contains them is not really an array in the traditional sense. What you need instead is a map between question numbers and the object that maintains the data structure for a particular question.

**Coding the program**

Once you have defined the internal data structure and the external file format, the process of writing the code for the teaching machine is surprisingly straightforward, as long as you decompose the entire task into simpler functions using stepwise refinement. The program is divided into three files: *TeachingMachine.js* is the main program, *TMCourse.js* defines the class that contains the data for a course, and *TMQuestion.js* defines the class that represents a single question. The code for these files appears in Figures 9-11, 9-12, and 9-13 on the next three pages.
/*
 * File: TeachingMachine.js
 * ________________
 * This program executes a programmed instruction course.
 */

function TeachingMachine() {
    function processFile(text) {
        let course = TMCourse(text);
        stepThroughCourse(course);
    }
    JSFile.chooseTextFile(processFile);
}

/* Steps through the course, asking questions and checking responses */

function stepThroughCourse(course) {
    let qnum = 1;

    function askQuestion() {
        if (qnum === 0) {
            console.log("Done");
        } else {
            let question = course.getQuestion(qnum);
            if (question === undefined) {
                console.log("Missing question " + qnum);
            } else {
                question.printQuestionText();
                console.requestInput("Answer: ", checkAnswer);
            }
        }
    }

    function checkAnswer(line) {
        let question = course.getQuestion(qnum);
        let nextQuestion = question.lookupAnswer(line);
        if (nextQuestion === undefined) {
            console.log("I don't understand that response.");
        } else {
            qnum = nextQuestion;
        }
        askQuestion();
    }

    console.log(course.getTitle());
    askQuestion();
}
/*
 * File: TMCourse.js
 * ---------------------
 * This class defines the data structure for a course for use with
 * the TeachingMachine program.
 */

/* Constants */

const MARKER = "----";

/*
 * Creates a new course for the teaching machine by reading the data from the
 * specified file, which consists of questions and their accepted answers.
 */

function TMCourse(text) {
    let lines = JSFile.convertToLineArray(text);
    let title = lines.shift();
    let questions = [ ];
    let line = lines.shift();
    while (line !== undefined) {
        let qnum = parseInt(line);
        let text = [ ];
        line = lines.shift();
        while (line !== undefined && line !== MARKER) {
            text.push(line);
            line = lines.shift();
        }
        let question = TMQuestion(text);
        line = lines.shift();
        while (line !== undefined && line !== "]") {
            let colon = line.indexOf(":");
            let response = line.substring(0, colon).toLowerCase().trim();
            let nextQuestion = parseInt(line.substring(colon + 1).trim());
            question.addAnswer(response, nextQuestion);
            line = lines.shift();
        }
        questions[qnum] = question;
        line = lines.shift();
    }
    return {
        getTitle: function() { return title; },
        getQuestion: function(qnum) { return questions[qnum]; }
    };
}
There are a few aspects of the code that are worth going over in more detail, but it is also important for you to start working your way through longer programs, even if you don’t understand all of the code at first. It also helps to look for familiar patterns in the code. The TeachingMachine function at the top of Figure 9-11 is simply an instance of the standard pattern for choosing a text file and then using a callback function to process its contents. The stepThroughCourse function divides its work into two inner functions, askQuestion and checkAnswer, that are called alternately as user goes through the course. If you look at these functions independently, you will find that each of them does precisely what you would expect.

The code for the TMCourse class in Figure 9-12 consists primarily of reading through the lines of a file and initializing the fields of the class from that data. The internal state for the TMCourse object is maintained in the local variables title and questions. The TMCourse class exports two methods—getTitle and getQuestion—that give clients access to the data. Similarly, the TMQuestion class in Figure 9-13 stores its data in the variables text and answerTable, which clients obtain by calling printQuestionText, addAnswer, and lookupAnswer.
Summary

This chapter introduces the concept of an object, which JavaScript uses in both the more traditional sense of a collection of values and its more modern sense as a structure that encapsulates data and behavior. Like arrays, objects make it possible to combine multiple values into a single unit. In an array, individual elements are selected by number; in an object, individual fields are selected by name.

The important points introduced in this chapter include:

- You can create an object in JavaScript by enclosing a series of name-value pairs inside curly braces. For example, the expression `{ name: "two", value: 2 }` defines an object whose name field is the string "two" and whose value field is the number 2. This format is called JavaScript Object Notation or JSON.
- Given a JavaScript object, you can select individual fields using the dot operator.
- Like arrays, objects are treated as references, which means that they are not copied when the object is assigned or passed as a parameter.
- Modern programming style discourages clients from manipulating the values of individual fields directly. Programs that follow the object-oriented model provide access to the data through the use of methods, which increase the conceptual separation between the client and the implementation. The client and the implementation come together through an interface, which gives the client the necessary level of access while hiding as many of the details as possible.
- This chapter implements inheritance through a hierarchical layering of the factory methods used to create new objects. The factory method for a subclass calls the factory method for its superclass and then adds any new definitions that that subclass requires.
- Objects in JavaScript implement an important conceptual structure called a map, which is a structure that provides an association between keys and values.
- Programs in which the control flow is determined by the data structure are said to be data driven. Data-driven programs are usually shorter, more flexible, and easier to maintain than programs that incorporate the same information directly into the program design.
- Data-driven programs typically have two formats for the data: an external format stored in a data file and an internal format stored as a hierarchical combination of objects, arrays, and built-in values.

Review questions

1. Who were the inventors of object-oriented programming?
2. True or false: JavaScript uses the term *object* to refer both to the older notion of a data aggregate and the more modern notion of a structure that encapsulates data and behavior.

3. What word does JavaScript use for the individual components of an object?

4. What does *JSON* stand for?

5. What is the *dot operator* and how is it used?

6. True or false: If you pass a JavaScript object as a parameter to a function, the function receives a copy of the object and therefore cannot change the fields of the original.

7. What would happen if you changed the value of *DELTA* in *YarnProgram.js* from 113 to 104? Would the picture be as striking? Why or why not?

8. True or false: Modern programming practice discourages direct access to the data fields in an object.

9. In your own words, describe the concept of an *interface*.

10. How do you add a method to an object?

11. How can you hide internal information from clients of a class?

12. What are *getters* and *setters*?

13. What is the special role of the *toString* method in an object?

14. How does a subclass extend the behavior of its superclass?

15. What is a *map*? How do you implement the map concept in JavaScript?

16. What are the advantages of *data-driven programs*?

---

### Exercises

1. Write a function *printPayroll* that takes an array of employees, each of which is defined as a simple JavaScript object using JSON notation, and prints on the console a list for each employee showing the name, title, and salary. For example, if *SCROOGE_AND_MARLEY* has been initialized as a two-element array containing the entries for Ebenezer Scrooge and Bob Cratchit shown in the chapter, your function should be able to generate the following sample run:
This implementation of `printPayroll` uses the `alignLeft` and `alignRight` functions from Chapter 5, exercise 2, to align the name and title at the left of a 25-character field and the salary at the right of a six-character field.

2. Complete the definition of the `Employee` hierarchy from Figures 9-5 and 9-6 by defining the classes `CommissionedEmployee` and `SalariedEmployee`.

3. Complete the implementation of the `DrawTrain.js` program in Figure 9-8 on page 312 by writing definitions for the `Engine` and `Caboose` classes. Update the main program so that the train includes an engine, a boxcar, and a caboose.

4. In Chapter 6, you had the chance to see and write several programs that allowed you to create shapes by dragging the mouse on the graphics window. Using those programs as starting point, create a more elaborate drawing program that displays an onscreen menu of five shapes—a filled rectangle, an outlined rectangle, a filled oval, an outlined oval, and a straight line—along the left side of the canvas, as shown in the following diagram:

Clicking on one of the squares in the menu chooses that shape as a drawing tool. Thus, if you click on the filled oval in the middle of the menu area, your program should draw filled ovals. Clicking and dragging outside the menu should draw the currently selected shape.

Each of the drawing tools along the left edge of a window should be a `GCompound` that combines the symbol for the tool and the enclosing square. Each tool, moreover, should define additional methods that create the shapes in response to the mouse events. Your program should therefore check to see if a mouse click is inside one of the tools and, if so, store that tool in a local variable so that the callback functions for the mouse events can perform whatever actions are required to implement that tool.
5. Extend the DrawShapes application from the preceding exercise so that the left sidebar also includes a palette of colors. Clicking on one of the colors sets the current color for the application, so that subsequent shapes are drawn in that color. There is not enough room to display all 140 of JavaScript’s predefined colors, so you will need to choose a subset that you find aesthetically pleasing.

6. Extend the DrawShapes application from exercises 4 and 5 so that clicking on an existing shape drags that shape around the screen. Your implementation should not allow the user to drag the shape and color tools, so you will need to check that the "mousedown" event that starts the drag operation has not occurred in one of these.

7. In Roman numerals, characters of the alphabet are used to represent integers as shown in this table:

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>V</td>
<td>5</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>50</td>
</tr>
<tr>
<td>C</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>500</td>
</tr>
<tr>
<td>M</td>
<td>1000</td>
</tr>
</tbody>
</table>

Each character in a Roman numeral stands for the corresponding value. Ordinarily, the value of the Roman numeral as a whole is the sum of the individual character values given in the table. Thus, the string "LXXVI" denotes $50 + 10 + 10 + 5 + 1$, or 76. The only exception occurs when a character corresponding to a smaller value precedes a character representing a larger one, in which case the value of the first letter is subtracted from the total, so that the string "IX" corresponds to $10 - 1$, or 9.

Write a function `romanToDecimal` that takes a string representing a Roman numeral and returns the corresponding decimal number. To find the values of each Roman numeral character, your function should look that character up in a map that implements the Roman numeral conversion. If the string contains characters that are not in the table, `romanToDecimal` should return -1.

8. In May of 1844, Samuel F. B. Morse sent the message “What hath God wrought!” by telegraph from Washington to Baltimore, heralding the beginning of the age of electronic communication. To make it possible to communicate information using only the presence or absence of a single tone, Morse designed a coding system in which letters and other symbols are represented as coded sequences of short and long tones, traditionally called dots and dashes. In Morse code, the 26 letters of the alphabet are represented by the codes shown in Figure 9-14 at the top of the next page.
Write a program that reads in lines from the user and translates each line either to or from Morse code, depending on the first character of the line:

- If the line starts with a letter, you want to translate it to Morse code. Any characters other than the 26 letters should simply be ignored.
- If the line starts with a period (dot) or a hyphen (dash), it should be read as a series of Morse code characters that you need to translate back to letters. You may assume that each sequence of dots and dashes in the input string will be separated by spaces, and you are free to ignore any other characters that appear. Because there is no encoding for the space between words, the characters of the translated message will be run together when your program translates in this direction.

The program should end when the user enters a blank line. A sample run of this program (taken from the messages between the Titanic and the Carpathia in 1912) might look like this:

9. A rational number is one that can be expressed as the quotient of two integers. For example, the number 1.25 is a rational number because it is equal to 5 divided by 4. Rational numbers show up in elementary-school arithmetic as fractions, which consist of a numerator and a denominator, both of which are integers.
Although rational numbers are not a built-in type in JavaScript, you can define a `Rational` class that implements computation using rational values. Each object of type `Rational` stores the numerator and denominator internally but does not make those values directly available to clients. Clients instead call methods on `Rational` objects to perform the basic arithmetic operations of addition, subtraction, multiplication, and division.

Using the `Point` class as a model, create a file called `Rational.js` that exports the following entries:

- A factory method that returns a `Rational` object. The factory method ordinarily takes two arguments, which are the numerator and denominator of the fraction. For example, calling `Rational(5, 4)` should return a `Rational` object corresponding to the rational number 1.25, or 5/4. The `Rational` factory method should also accept a single argument, in which case, the denominator should be set to 1, so that `Rational(2)` corresponds to the integer 2. Your implementation of `Rational` should reduce the fraction to lowest terms by dividing both the numerator and the denominator by their greatest common divisor.

- The methods `add`, `sub`, `mul`, and `div`, which implement the standard arithmetic operators using the formulas shown in Figure 9-15. Keep in mind that these operations are implemented as methods, which means that they use the receiver syntax. Thus, to express the computation

  \[
  \frac{1}{2} + \frac{1}{3} + \frac{1}{6}
  \]

  you would need to write the following JavaScript expression:

  ```javascript
  Rational(1, 2).add(Rational(1, 3)).add(Rational(1, 6))
  ```

- A `toString` method that converts the `Rational` object to a string consisting of the numerator and the denominator separated by a slash. Thus, calling `Rational(5, 4).toString()` should produce the string "5/4".

---

**Figure 9-15** Rules for rational arithmetic

<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>( \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd} )</td>
</tr>
<tr>
<td>Subtraction</td>
<td>( \frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd} )</td>
</tr>
<tr>
<td>Multiplication</td>
<td>( \frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd} )</td>
</tr>
<tr>
<td>Division</td>
<td>( \frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc} )</td>
</tr>
</tbody>
</table>
10. Suppose that you have been hired as a programmer for a bank to automate the process of converting between different foreign currencies at the prevailing rate of exchange. Every day, the bank receives a file containing the exchange rates for that day. Each line of the file consists of the name of the currency, the three-letter code for that currency enclosed in parentheses, and that day’s exchange rate relative to the dollar. For example, the line

| Pound sterling (GBP) | 1.23586 |

Indicates that the British pound has the name "Pound sterling", the symbol "GBP" and is currently trading at 1.23586 dollars to the pound. Figure 9-16 shows the first 15 lines of the data file for September 13, 2017.

Your task in this problem is to write a program that allows the user to select the data file, reads the contents of that file into a suitable internal structure, and then accepts conversion requests from the user, which are either just the three-letter code for a currency, in which case the program should display the currency name, or a line in the form:

\[
amount \ XXX \rightarrow \ YYY
\]

where \(amount\) is the monetary value you want to convert, \(XXX\) and \(YYY\) are the three-letter codes for the old and new currency, and the spaces shown in the line are optional. A sample run of your program might look like this:

**Figure 9-16** Beginning of a data file containing exchange rates for a specific day

<table>
<thead>
<tr>
<th>Exchange Rates-13-Sep-2017.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>US Dollar (USD)</td>
</tr>
<tr>
<td>Euro (EUR)</td>
</tr>
<tr>
<td>British Pound (GBP)</td>
</tr>
<tr>
<td>Indian Rupee (INR)</td>
</tr>
<tr>
<td>Australian Dollar (AUD)</td>
</tr>
<tr>
<td>Canadian Dollar (CAD)</td>
</tr>
<tr>
<td>Singapore Dollar (SGD)</td>
</tr>
<tr>
<td>Swiss Franc (CHF)</td>
</tr>
<tr>
<td>Malaysian Ringgit (MYR)</td>
</tr>
<tr>
<td>Japanese Yen (JPY)</td>
</tr>
<tr>
<td>Chinese Yuan Renminbi (CNY)</td>
</tr>
<tr>
<td>New Zealand Dollar (NZD)</td>
</tr>
<tr>
<td>Thai Baht (THB)</td>
</tr>
<tr>
<td>Hungarian Forint (HUF)</td>
</tr>
<tr>
<td>Emirati Dirham (AED)</td>
</tr>
</tbody>
</table>
In J. K. Rowling’s *Harry Potter* series, the students at Hogwarts School of Witchcraft and Wizardry study many forms of magic. One of the most difficult fields of study is potions, which is taught by Harry’s least favorite teacher, Professor Snape (played in the movies by the late Alan Rickman). Mastery of potions requires students to memorize complex lists of ingredients for creating the desired magical concoctions. Presumably to protect those of us in the Muggle world, Rowling does not give us a complete ingredient list for most of the potions used in the series, but we do learn about a few, including those shown in the following data file:

```
JavaScript Console
Enter requests in the form nXXX -> YYY, where n is the amount you want to convert and XXX and YYY are the three-letter currency codes. Entering only a currency code gives the name of that currency.

Conversion: 1.00USD -> JPY
1 USD = 110.11441294928758 JPY

Conversion: 200 GBP -> EUR
200 GBP = 221.82879596017673 EUR

Conversion: CHF
CHF = Swiss Franc
Conversion:
```

```
11. In J. K. Rowling’s *Harry Potter* series, the students at Hogwarts School of Witchcraft and Wizardry study many forms of magic. One of the most difficult fields of study is potions, which is taught by Harry’s least favorite teacher, Professor Snape (played in the movies by the late Alan Rickman). Mastery of potions requires students to memorize complex lists of ingredients for creating the desired magical concoctions. Presumably to protect those of us in the Muggle world, Rowling does not give us a complete ingredient list for most of the potions used in the series, but we do learn about a few, including those shown in the following data file:

```
Potions.txt
Polyjuice Potion
shredded boomslang skin
lacewing flies
leeches
knotgrass
powdered bicorn horn
fluxweed
a bit of the person one wants to become

Wit-Sharpening Potion
ground scarab beetle
ginger root
armadillo bile

Shrinking Solution
chopped daisy roots
skinned shrivelfig
sliced caterpillar
rat spleen
leech juice

Draught of Living Death
asphodel in an infusion of wormwood
valerian roots
soporific bean
```
As the example illustrates, the format of the data file is a sequence of individual potions, each of which consists of the name of the potion, followed by a list of ingredients, one per line. Each of the potions is separated from the next by a blank line. Your task in this problem is to design the data structures necessary to represent the information about potions contained in this file, which requires you to complete the following tasks:

a) **Design a class to represent an individual potion.** Your first step in this process is to define a class called `Potion` that represents a single potion, which has a name and a list of ingredients. Your class should implement the following public interface:

- A factory method that takes the name of the potion and creates a `Potion` object with that name and an empty list of ingredients.
- A `getName` method that returns the name of the potion.
- An `addIngredient` method that adds an ingredient to the `Potion`.
- A `getIngredients` method that returns an array of the ingredients required for the potion listed in the order in which they were added.

b) **Design a class to represent a collection of potions.** Your next step is to define a `PotionCollection` class that represents the entire collection of potions. This class should export the following entries:

- A factory method that takes the text of a data formatted in the style described on the previous page and creates the corresponding potion collection.
- A `getPotion` method that takes the name of a potion and returns the `Potion` object that has that name, or `undefined` if there is no potion with that name.
- A `getPotionNames` method that returns an array list of the potion names in the order in which they appeared in the file.

c) **Write a main program that tests your implementation of the data structure.** Your program should ask the user to choose a data file containing the collection of potions, ask the user for a potion name, and then list all the necessary ingredients on the console.
Index

- operator, 42
-- operator, 46
-= operator, 46
! operator, 106
!== operator, 105, 227
. operator, 298
* operator, 42
/ operator, 42
& operator, 106
% operator, 42
+ operator, 42, 228
++ operator, 46
+= operator, 45, 228
< operator, 105, 227
<= operator, 105, 227
=== operator, 105, 227
> operator, 105, 227
>= operator, 105, 227
|| operator, 106

abs function, 52
abstract class, 87
abstraction, 148
acronym, 236
add method, 76
addEdge method, 204
addEventListener method, 186, 277
AddIntegerList.js, 243

addition, 42
addPolarEdge method, 204
addVertex method, 204
aggregate, 296
airport codes, 314
al-Khwarizmī, Muhammad ibn Mūsā, 23
algorithm, 21, 126
alignCenter function, 175
alignLeft function, 175
alignRight function, 118, 175
AnimatedSquare.js, 196
animation, 194
applying a function, 183
architect, 90
argument, 49, 144, 159
arithmetic expression, 42
array, 256
array index, 256
array selection, 257
ascent, 80
ASCII, 226
assignment, 45
assignment statement, 45
Association for Computing Machinery, 181
associativity, 43
asynchronous interaction, 182
atan function, 52
atan2 function, 52
attribute, 66
Babbage, Charles, 224
Baby, 115
Babylonian method, 127
BabylonianSquareRoot.js, 129
backslash, 226
Bacon, Francis, 225
Bacon’s cipher, 225
baseline, 80
Baudot, Émile, 225
Baudot code, 225
beeper, 3
beepersInBag function, 11
beepersPresent function, 11
Berners-Lee, Tim, 63
biological classification system, 85
biological hierarchy, 84
bit, 220
BlueRectangle.js, 73
body tag, 67
Boole, George, 105
Boolean expressions, 104
break statement, 112
Breakout game, 216
Bricklin, Dan, 255
browser, 65
brute-force approach, 131
bug, 15
byte, 221
C(n, k), 160
Caesar, Julius, 241
Caesar cipher, 241
callback function, 186
calling a function, 3
camel case, 48
Čapek, Karel, 2
capitalize function, 247
cardinal number, 249
cascading if statement, 110
Cascading Style Sheets, 66
case clause, 112
ceil function, 52
celsiusToFahrenheit function, 50
Centering labels, 81
centimetersToFeetAndInches, 56
charAt method, 228, 229, 233
charCodeAt method, 228, 241
closure, 183
code, 3
coding, 90
collision model, 83
CollectBeeperTowers.k, 21
combinations function, 160
Combinations.js, 161
comment, 7
Conan Doyle, Arthur, 92
concat method, 262
concatenation, 54, 227
concrete class, 87
conditional statement, 10
console object, 243
cos function, 52
CountLetterFrequencies.js, 272
CPSR, 143
Craps.js, 156
crudeArray function, 263, 264
craceCrossedBox function, 207
createFilledCircle function, 87
createRGBPixel method, 284
createStar function, 206
Crockford, Douglas, 39, 63
cryptogram, 271
internal data structure, 319
timeout function, 195
inverting a key, 253
isDigit function, 231
isEven function, 147
isLeapYear function, 148
isNaN function, 243
isPalindrome function, 235
isVisible method, 188
iterative statements, 13

James, P. D., 92
JavaScript, 40, 92
JavaScript colors, 75
JavaScript console, 42
JavaScript coordinate system, 74
JavaScript Object Notation, 297
JPEG format, 277
JSConsole.js library, 68
JSFile.js library, 267
JSGraphics.js library, 69
JSON, 297

Karel, 2
Karel console, 4
Kay, Alan, 181
key, 314

Lasius niger, 85
lastIndexOf method, 228, 262
left-associative, 43
leftIsBlocked function, 11
leftIsClear function, 11
Lehrer, Tom, 42
Leibniz, 220
length field, 228, 229, 257
let keyword, 45
letter-substitution cipher, 252
lexicographic order, 227
library, 8, 51, 148, 303
library functions, 51
Linnaeus, Carl, 84
listArray function, 258
load event, 277
local variable, 50, 159
log function, 52
log10 function, 52
logical and, 106
logical not, 106
logical operator, 106
logical or, 106
LOGO, 1
Lord of the Rings, 227
Lovelace, Ada, 224
luminance, 284

magic square, 290
Manchester University, 115
map, 314
Math library, 52
Math.abs function, 52
Math.atan function, 52
Math.atan2 function, 52
Math.ceil function, 52
Math.cos function, 52
Math.E constant, 52
Math.exp function, 52
Math.floor function, 52, 116
Math.log function, 52
Math.log10 function, 52
Math.max function, 52
Math.min function, 52
Math.PI constant, 52
Math.pow function, 52
Math.random function, 52, 151
Math.round function, 52
Math.sin function, 52
Math.tan function, 52
matrix, 275
max function, 52
mean, 288
Menabrea, Luigi, 224
message, 72
method, 72
microworld, 2
min function, 52
Mindstorms, 1
Minesweeper game, 292
model, 82
monospaced font, 79
Monte Carlo integration, 177
monthName function, 150
MOOC, 317
Morse, Samuel F. B., 329
Morse code, 329
mouse event, 187
mousedown event, 189
mousetmove event, 189
mouseup event, 189
move function, 4
move method, 188
MoveBeeperToLedge.k, 7
movePolar method, 188
muggle, 333
multidimensional array, 275
multiplication, 42
NaN constant, 243
nCopies function, 234
negation, 42
nested control statements, 13
nested for statements, 123
newline character, 266
Newspeak, 61
noBeepersInBag function, 11
noBeepersPresent function, 11
nondeterminism, 151
nonnumeric data, 224
notFacingEast function, 11
notFacingNorth function, 11
notFacingSouth function, 11
notFacingWest function, 11
null constant, 193
number cruncher, 40
numbers, 41
numerator, 330
numeric data, 40
Nygaaard, Kristen, 295
Obenglobish, 250
object, 71, 296
object-oriented paradigm, 72, 182
object-oriented programming, 295
one-shot timer, 195
onload attribute, 68
OOP, 182
operator, 42
ordinal number, 249
origin, 80
Orwell, George, 61
overriding, 310
PacMan, 215
palindrome, 235
Papert, Seymour, 1
parameter, 49, 145, 159
parameter passing, 145
Parnas, David, 143
parseFloat function, 243
parseInt function, 223, 243
Pascal, Blaise, 178
Pascal’s Triangle, 178
pattern, 20
perfect number, 141, 246
permutation, 178
PI constant, 48, 52
pickBeeper function, 4
Pig Latin, 237
PigLatin.js, 238
pixel, 70
plot function, 211
PNG format, 277
Poe, Edgar Allan, 271
point, 299
PointClass.js, 307
polar coordinates, 204
Polybius square, 240
designed, 203
pop method, 262
postcondition, 21
Potter, Harry, 216
pow function, 52
precedence, 43
precondition, 21
predicate function, 147
prime number, 141
printFunctionTable function, 183
problem-solving, 4
procedure, 145
program, 3
programmed instruction, 317
programming, 2
programming abstraction, 148
programming idiom, 20
programming pattern, 20
promise, 283
prompt, 243
property, 229
pseudocode, 116
push method, 262, 263
putBeeper function, 4

quadratic formula, 93
question-mark colon, 111

R.U.R., 2
radioactive decay, 175
rainbow, 101
random function, 52, 151
randomChance function, 153, 155
RandomCircles.js, 157
randomColor function, 153, 155
randomInteger function, 152, 153
RandomLib.js, 153
randomReal function, 153, 155
rational number, 330
receiver, 72
receiver syntax, 72, 227
reductionism, 144
reference, 70, 260
reference point, 203
relational operator, 105
remainder, 42
repeat statement, 13
replace method, 228
requestInput method, 243
reserved word, 47
return address, 159
return statement, 49, 159
returning a result, 49
reverse function, 234
reverse method, 262
reverseArray function, 259
Rich Pattis, 2
Rickman, Alan, 333
right-associative, 43
right-hand rule, 23
rightIsBlocked function, 11
rightIsClear function, 11
robot, 2
Rossum's Universal Robots, 2
rotate method, 188
round function, 52
Rowling, J. K., 216, 254, 257, 333
row-major order, 276
rubber-banding, 190
sans-serif font, 79

scale method, 188
scientific notation, 41
scope, 197
Scrabble, 247
script tag, 67
selection, 257
sendBackward method, 188
sendForward method, 188
sendToBack method, 188
sendToFront method, 188
serif font, 79
set of operations, 40
setBounds method, 188
setColor method, 75, 188
setEndPoint, 188
setFillColor method, 188, 202, 205
setFilled method, 188, 202, 205
setInterval function, 195
setLineWidth method, 188
setLocation method, 188
setStartAngle method, 202
setStartPoint method, 188
setSweepAngle method, 202
setter, 305
setTimeout function, 195
setVisible method, 188
Shakespeare, William, 240, 265
Shaw, George Bernard, 249
Sherlock Holmes, 92
shift method, 262
short-circuit evaluation, 108
shorthand assignment, 46
SIMULA, 295
sin function, 52
slice method, 262
Small-Scale Experimental Machine, 115
Smith, Alexander McCall, 92
solving a maze, 23
sort method, 262
speed of light, 41
splice method, 262
split method, 228, 269
square root, 127
St. Ives, 60
stack frame, 159
stacking order, 83
standard deviation, 288
Stanford Graphics Library, 69
start angle, 200
starts method, 228, 232
StatsLib.js, 288
Stephenson, Neal, 242
stepwise refinement, 18, 166
Stoppard, Tom, 177
string, 53
string data, 40
string operations, 54
Stroop, J. Ridley, 180
structure, 296
subclass, 87, 308
subproblem, 8
substring, 230
substring method, 228, 230
subtraction, 42
successive approximation, 127
Sudoku, 293
Suetonius, 241
superclass, 87, 308
Swade, Doron, 225
sweep angle, 200
switch statement, 112
synchronous interaction, 182
syntactic rules, 3

tag, 66
tan function, 52
Target.js, 87
TeachingMachine.js, 323
Tengwar, 227
term, 42
testing, 90, 94
text editor, 65
text file, 265
Tic-Tac-Toe, 100, 275
time step, 195
timer, 195
title tag, 67
TMCourse.js, 324
Tolkien, J. R. R., 227, 287
toLowerCase method, 228, 231
top-down design, 18, 166
toPigLatin function, 238
toString method, 224, 307
toUpperCase method, 228, 231
Train class, 311
TrainCar class, 311
trim method, 232
true constant, 105
truth table, 106
truthy, 134
Turing, Alan, 115
Turing award, 63, 115, 295
turnAround function, 8
turnLeft function, 4
turnRight function, 6
turns library, 8

UML, 86
UML diagram, 86, 308
undefined constant, 159, 257
Unicode, 227
universal modeling language, 86
unshift method, 262
URL, 66

value, 314
vandal, 90
variable, 44
Venerable Bede, 142
Vermeer, Jan, 294
vertex, 203
VisiCalc, 255

web, 65
web browser, 65
while loop, 114
while statement, 14, 114
Wiener Award, 143
Wilkes, Maurice, 89
word, 221
wordToPigLatin function, 238
World Wide Web, 65

Xerox PARC, 242

YarnPattern.js, 301

z-axis, 83
z-order, 83