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:
>
> *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:

```
function HelloWorld() {
   console.log("hello, world");
}
```

FIGURE 3-1 The "Hello World" program in 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
 * book, The C Programming Language.
 */

function HelloWorld() {
   console.log("hello, world");
}
```

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

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 need 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

```
<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

```
<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:

```
<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:

```
<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:

```
<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"

```
<!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="JSGraphice.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**

```
/*
 * 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
 * Brian Kernighan and Dennis Ritchie's book, The C Programming Language.
 */

/* 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);
}
```

FIGURE 3-4 The `index.html` file for `GraphicsHelloWorld.js`

```
<!DOCTYPE html>
<html>
  <head>
    <title>GraphicsHelloWorld</title>
    <script type="text/javascript" src="JSGraphics.js"></script>
    <script type="text/javascript" src="GraphicsHelloWorld.js"></script>
  </head>
  <body onload="GraphicsHelloWorld()"></body>
</html>
```

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

```
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

```
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.

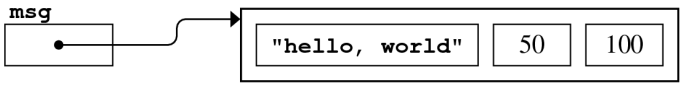
In JavaScript, a value that represents some usually larger and more complex value is called a ***reference.*** In this case, the variable **`gw`** is initialized to contain a reference to a portion of the browser window capable of displaying graphical objects, as illustrated by the following diagram:

As the arrow suggests, the reference stored in **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 ***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 ***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 ***instance*** of that class.

The second line in the function

```
let msg = GLabel("hello, world", 50, 100);
```

operates in a similar fashion. This line creates a **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 **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 **GLabel** but are instead securely packaged inside the object. This model of packaging together data and code is called ***encapsulation.***

Even though the declarations of the variables **gw** and **msg** create the necessary objects, these lines alone do not cause the **GLabel** to appear in the **GWindow**. To get the message to appear, the program has to tell the **GWindow** object stored in **gw** to add the **GLabel** stored in **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 **GraphicsHelloWorld.js** program, which looks like this:

```
gw.add(msg);
```

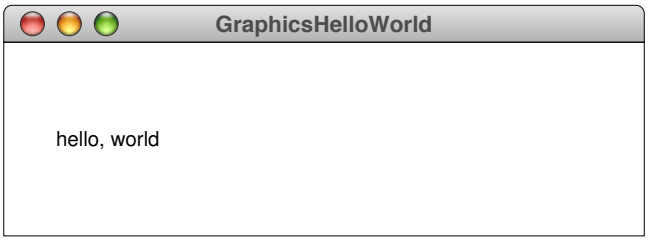
Understanding how this statement works requires you to understand a little more about the way that JavaScript works with objects.

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:

```
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:

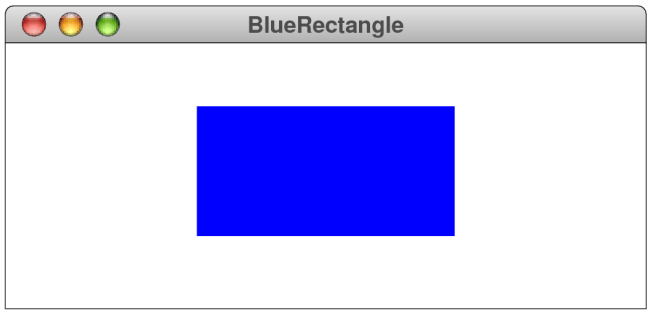


FIGURE 3-5 Program to draw a blue rectangle on the graphics window

```
/*
 * 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**.

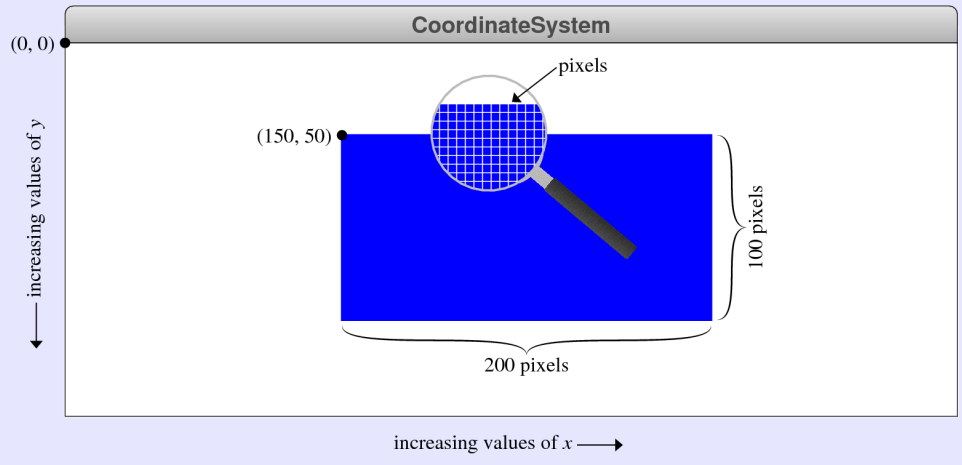
The next statement in the **BlueRectangle** function is

```
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

FIGURE 3-6 The coordinate system used 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

AliceBlue	DarkSlateGrey	LightPink	PaleVioletRed
AntiqueWhite	DarkTurquoise	LightSalmon	PapayaWhip
Aqua	DarkViolet	LightSeaGreen	PeachPuff
Aquamarine	DeepPink	LightSkyBlue	Peru
Azure	DeepSkyBlue	LightSlateGray	Pink
Beige	DimGray	LightSlateGrey	Plum
Bisque	DimGrey	LightSteelBlue	PowderBlue
Black	DodgerBlue	LightYellow	Purple
BlanchedAlmond	FireBrick	Lime	RebeccaPurple
Blue	FloralWhite	LimeGreen	Red
BlueViolet	ForestGreen	Linen	RosyBrown
Brown	Fuchsia	Magenta	RoyalBlue
BurlyWood	Gainsboro	Maroon	SaddleBrown
CadetBlue	GhostWhite	MediumAquaMarine	Salmon
Chartreuse	Gold	MediumBlue	SandyBrown
Chocolate	GoldenRod	MediumOrchid	SeaGreen
Coral	Gray	MediumPurple	SeaShell
CornflowerBlue	Grey	MediumSeaGreen	Sienna
Cornsilk	Green	MediumSlateBlue	Silver
Crimson	GreenYellow	MediumSpringGreen	SkyBlue
Cyan	HoneyDew	MediumTurquoise	SlateBlue
DarkBlue	HotPink	MediumVioletRed	SlateGray
DarkCyan	IndianRed	MidnightBlue	SlateGrey
DarkGoldenRod	Indigo	MintCream	Snow
DarkGray	Ivory	MistyRose	SpringGreen
DarkGrey	Khaki	Moccasin	SteelBlue
DarkGreen	Lavender	NavajoWhite	Tan
DarkKhaki	LavenderBlush	Navy	Teal
DarkMagenta	LawnGreen	OldLace	Thistle
DarkOliveGreen	LemonChiffon	Olive	Tomato
DarkOrange	LightBlue	OliveDrab	Turquoise
DarkOrchid	LightCoral	Orange	Violet
DarkRed	LightCyan	OrangeRed	Wheat
DarkSalmon	LightGoldenRodYellow	Orchid	White
DarkSeaGreen	LightGray	PaleGoldenRod	WhiteSmoke
DarkSlateBlue	LightGrey	PaleGreen	Yellow
DarkSlateGray	LightGreen	PaleTurquoise	YellowGreen

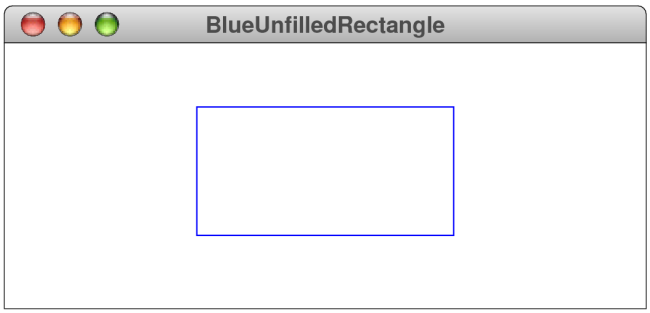
from designing web pages. If not, you will have a chance to learn more about hexadecimal notation in Chapter 7.

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

```
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 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:



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

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

```
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

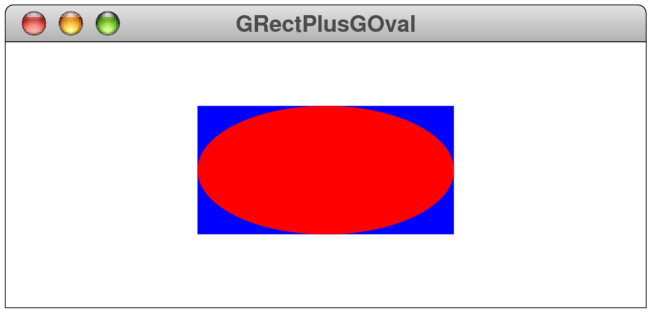
As its name suggests, the **GOval** class is used to display an oval-shaped figure in a graphics window. Structurally, the **GOval** class is similar to the **GRect** class: the **GOval** function itself takes the same arguments as the **GRect** function, and the two classes respond to the same set of 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:

```
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

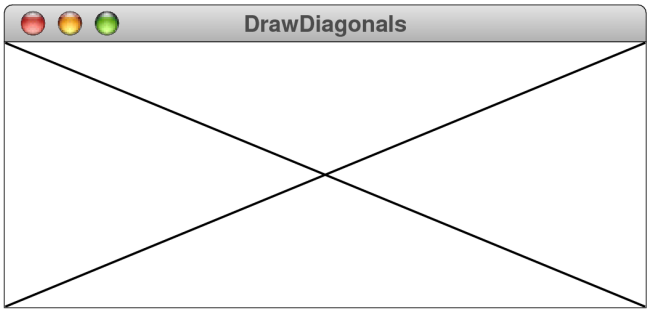
```
GLine(0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT)
```

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:

```
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:



The GLabel class

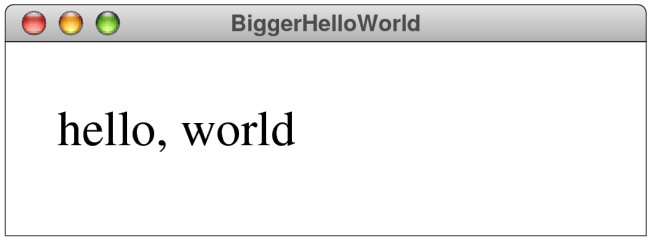
When you last saw the **GLabel** class in the **GraphicsHelloWorld.js** program, the results were not entirely satisfying. The message appeared on the screen, but was too small to generate much excitement. To make the **"hello, world"** message bigger, you need to display the **GLabel** in a different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

```
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:



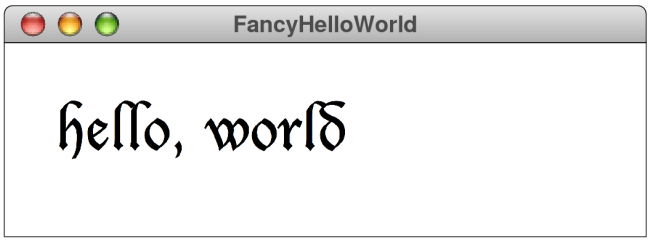
The string passed as the argument to **`setFont`** is written using CSS, which, as noted earlier in this chapter, is the technology the web uses to specify the visual appearance of the page. This string specifies several properties of the font, which appear in the following order:

- 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 **`monospaced`**, 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

```
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:

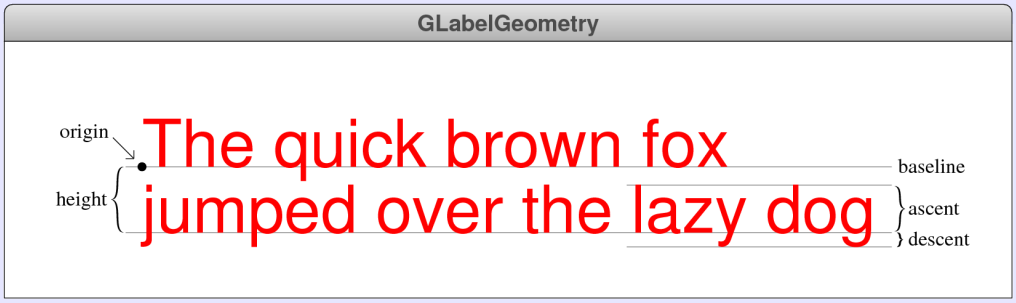


The **GLabel** class uses its own geometric model, which is similar to the ones that typesetters have used over the centuries since Gutenberg's invention of the printing press. The notion of a font, of course, originally comes from printing. Printers would load different sizes and styles of type into 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.

FIGURE 3-8 The geometry of the `GLabel` class



The **`GLabel`** class includes methods that allow you to determine these properties. For example, the **`GLabel`** class includes a method called **`getAscent`** to determine the ascent of the font in which the label appears. In addition, it includes a method called **`getWidth`** that determines the horizontal extent of the **`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 **`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 **`GLabel`** comes in two forms. The first takes the string for the label along with the *x* and *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 **`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 **`add`** method, which takes optional *x* and *y* parameters to set the origin of the object when you add it to the **`GWindow`**. The following program implements this strategy:

```
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 **`GLabel`** occur in the declarations of the variables **`x`** and **`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:

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

```
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.

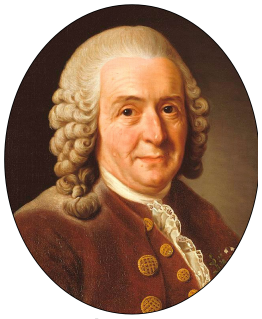
Collages also have the property that some objects can be positioned on top of other objects, obscuring whatever is behind them. Removing those objects reveals whatever used to be underneath. The back-to-front ordering of objects in the collage is called the ***stacking order*** in this book, although 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.

FIGURE 3-9 **Selected methods from the Stanford Graphics Library**

GWindow(*width***,** *height***)**	Creates a new **GWindow** object of the specified size.
*gw***.add(***obj***)**	Adds the object to the graphics window at its internally stored location.
*gw***.add(***obj***,** *x***,** *y***)**	Adds the object to the graphics window so that its origin is positioned at the point (*x*, *y*).
*gw***.remove(***obj***)**	Removes the object from the graphics window.
*gw***.getWidth()**	Returns the width of the graphics window.
*gw***.getHeight()**	Returns the height of the graphics window.

3.4 Class hierarchies

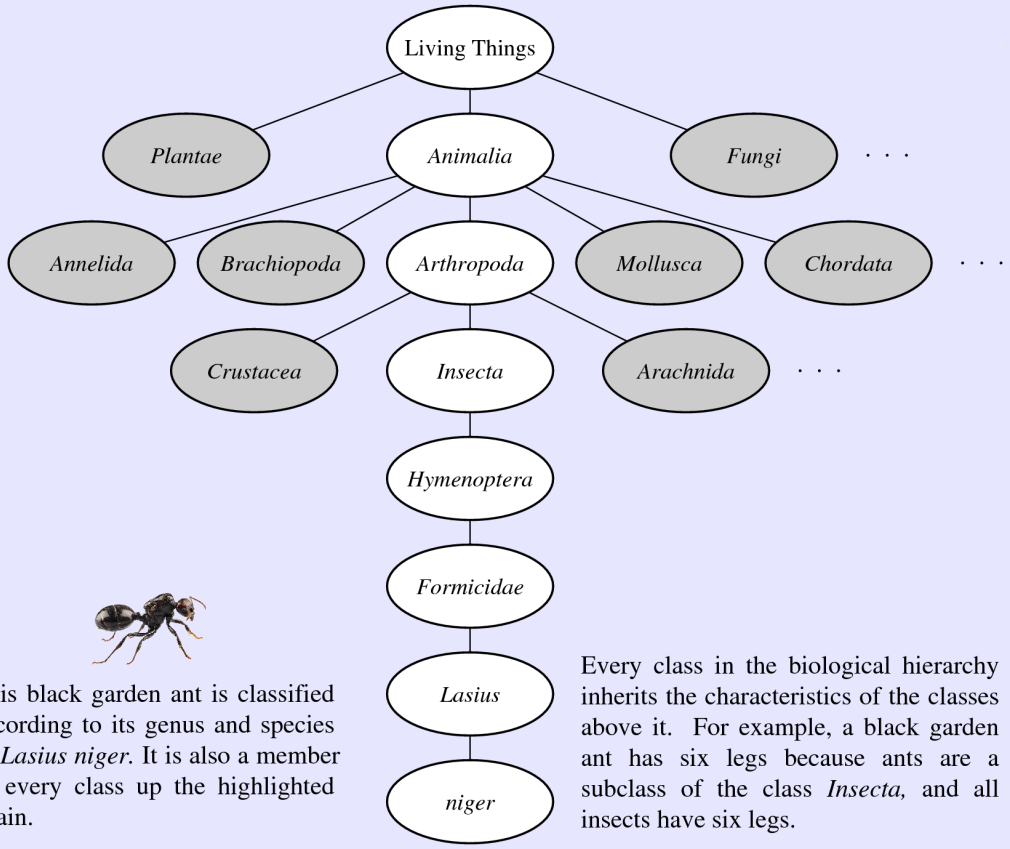


Carl Linnaeus

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,

FIGURE 3-10 **Class hierarchies in the biological world**



This black garden ant is classified according to its genus and species as *Lasius niger*. It is also a member of every class up the highlighted chain.

Every class in the biological hierarchy inherits the characteristics of the classes above it. For example, a black garden ant has six legs because ants are a subclass of the class *Insecta,* and all insects have six legs.

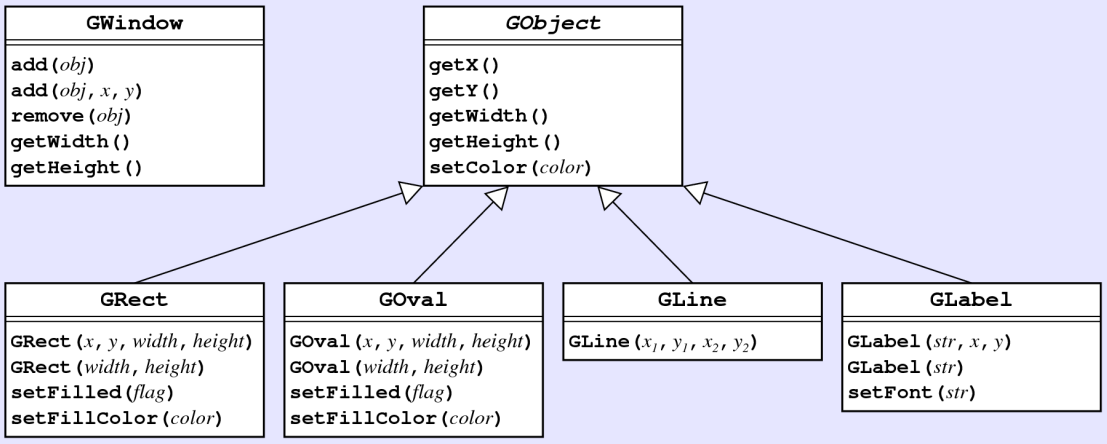
Lasius niger, corresponds to its genus and species. This species of ant, however, is also part of the family *Formicidae,* which is the classification that actually identifies it as an ant. If you move upward in the hierarchy from there, you discover that *Lasius niger* is also of the order *Hymenoptera* (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



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

GRect(*x*, *y*, *width*, *height*)	Creates a **GRect** object with the specified dimensions.
GRect(*width*, *height*)	Creates a **GRect** object of the specified size with its origin at (0, 0).
GOval(*x*, *y*, *width*, *height*)	Creates a **GOval** that fits inside the bounds of the corresponding rectangle.
GOval(*width*, *height*)	Creates a **GOval** object in which the oval fits inside a rectangle of the specified size. The origin of the **GOval** is (0, 0).
GLine(x_1, y_1, x_2, y_2)	Creates a **GLine** object connecting (x_1, y_1) and (x_2, y_2).
GLabel(*str*, *x*, *y*)	Creates a **GLabel** object containing the specified string with its baseline origin at the point (*x*, *y*).
GLabel(*str*)	Creates a **GLabel** object containing the specified string with its baseline origin at the point (0, 0).

Methods common to all graphical objects

object.**getX**()	Returns the *x* coordinate of the object.
object.**getY**()	Returns the *y* coordinate of the object.
object.**getWidth**()	Returns the width of the graphical object.
object.**getHeight**()	Returns the height of the graphical object.
object.**setColor**(*color*)	Sets the color of the object to *color*.

Methods available only for the GRect and GOval classes

object.**setFilled**(*flag*)	Sets whether this object is filled.
object.**setFillColor**(*color*)	Sets the color used to fill the interior of the object.

Methods available only for the GLabel class

object.**setFont**(*str*)	Sets the font for the label. The format of the font specification is a CSS string as described in the text.
object.**getAscent**()	Gets the ascent (maximum distance above the baseline) for the label font.
object.**getDescent**()	Gets the descent (maximum distance below the baseline) for the label font.

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:

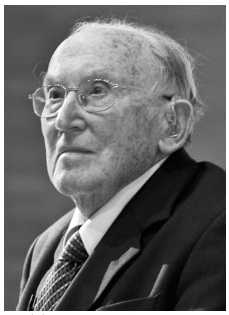
FIGURE 3-13 Program to draw a red and white target on the graphics window

```
/*
 * File: Target.js
 * ---------------
 * This program draws a target at the center of the graphics window
 * consisting of three concentric circles alternately colored red
 * and white.  The radius of the outer circle is specified by the
 * constant TARGET_RADIUS; the inner circles are two-thirds and
 * one-third that size.
 */

/* Constants */

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 200;
const TARGET_RADIUS = 75;

/* Main program */

function Target() {
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   let xc = gw.getWidth() / 2;
   let yc = gw.getHeight() / 2;
   gw.add(createFilledCircle(xc, yc, TARGET_RADIUS, "Red"));
   gw.add(createFilledCircle(xc, yc, 2 * TARGET_RADIUS / 3, "White"));
   gw.add(createFilledCircle(xc, yc, TARGET_RADIUS / 3, "Red"));
}

/*
 * Creates a circle of radius r centered at the point (x, y) filled
 * with the specified color and returns the initialized GOval to
 * the caller.
 */

function createFilledCircle(x, y, r, color) {
   let circle = GOval(x - r, y - r, 2 * r, 2 * r);
   circle.setColor(color);
   circle.setFilled(true);
   return circle;
}
```

3.6 Testing and debugging

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.

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:


Maurice Wilkes

> 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.

Phases and roles in the programming process

Design = Architect
Coding = Engineer
Testing = Vandal
Debugging = Detective

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

```
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

```
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.

FIGURE 3-14 Debugging advice from detective fiction

Regard with distrust all circumstances which seem to favor our secret desires.
—Émile Gaboriau, *Monsieur Lecoq,* 1868

There is nothing like first-hand evidence.
—Sir Arthur Conan Doyle, *A Study in Scarlet,* 1888

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.
—Sir Arthur Conan Doyle, *A Scandal in Bohenia,* 1892

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.
—Sir Arthur Conan Doyle, *The Adventure of the Reigate Squires,* 1892

With method and logic one can accomplish anything.
—Agatha Christie, *Poirot Investigates,* 1924

Detection requires a patient persistence which amounts to obstinacy.
—P. D. James, *An Unsuitable Job for a Woman,* 1972

It was always more difficult than you thought it would be.
—Alexander McCall Smith, *The No. 1 Ladies' Detective Agency,* 1998

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 ± 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:

```
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 parenthesized 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
> 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:

```
Quadratic
> 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
>
```

FIGURE 3-15 Implementation of the `quadratic` function and an associated test program

```
/*
 * 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);
}
```

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

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:

  `<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:

  receiver`.`*name*`(`*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:

  ```
  let gw = 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:

```
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?

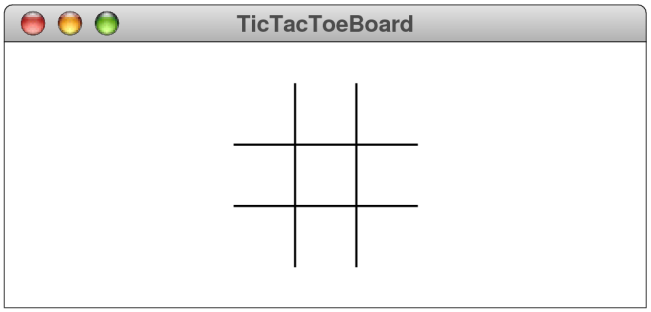
28. What built-in JavaScript function does the chapter identify as the most useful debugging tool?

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.

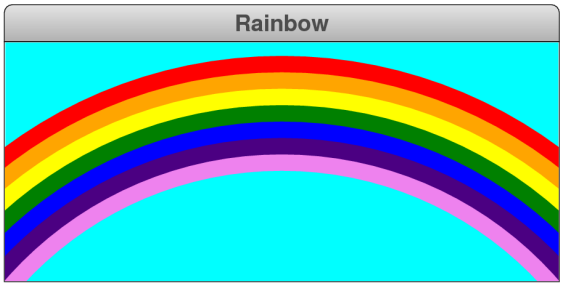
2. Do the same for the **`GraphicsHelloWorld.js`** program and the associated **`index.html`** from Figures 3-3 and 3-4.

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:



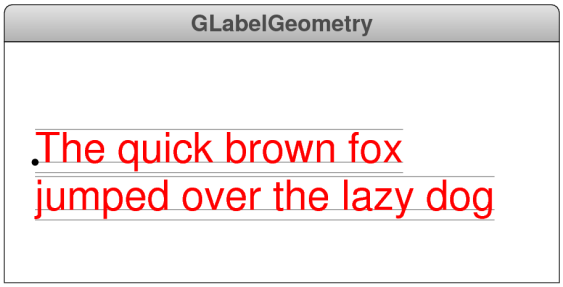
The size of the board should be specified as a constant, and the diagram should be centered in the window, both horizontally and vertically.

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



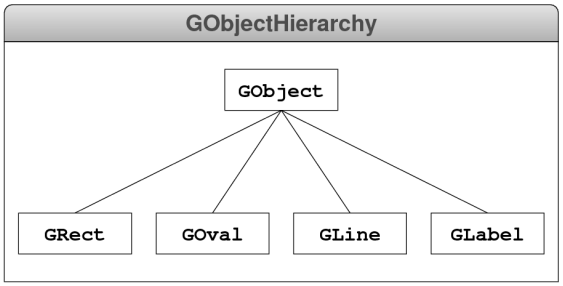
Starting at the top, the seven bands in the rainbow are red, orange, yellow, green, blue, indigo, and violet, respectively; cyan makes a lovely color for the sky. Remember that this chapter defines only the **`GRect`**, **`GOval`**, **`GLine`**, and **`GLabel`** classes and does not include a graphical 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:



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.