23.8 *(Blocked State)* List the reasons for entering the *blocked* state. For each of these, describe how the program will normally leave the *blocked* state and enter the *runnable* state.

23.9 *(Deadlock and Indefinite Postponement)* Two problems that can occur in systems that allow threads to wait are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time. Give an example of how each of these problems can occur in multithreaded Java programs.

23.10 *(Bouncing Ball)* Write a program that uses the JavaFX threading techniques introduced in this chapter to bounce a blue ball inside a `Pane`. The ball should begin moving in a random direction from the point where the user clicks the mouse. When the ball hits the edge of the `Pane`, it should bounce off the edge and continue in the opposite direction.

23.11 *(Bouncing Balls)* Modify the program in Exercise 23.10 to add a new ball each time the user clicks the mouse. Provide for a minimum of 20 balls. Randomly choose the color for each new ball.

23.12 *(Bouncing Balls with Shadows)* Modify the program in Exercise 23.11 to add shadows. As a ball moves, draw a solid black oval at the bottom of the `Pane`. You may consider adding a 3-D effect by increasing or decreasing the size of each ball when it hits the edge of the `Pane`.

23.13 Compare the use of synchronization vis-à-vis `ReentrantLock` for controlling access to shared objects.

23.14 *(Bounded Buffer: A Real-World Example)* Describe how a highway off-ramp onto a local road is a good example of a producer/consumer relationship with a bounded buffer. In particular, discuss how the designers might choose the size of the off-ramp.

Parallel Streams

For Exercises 23.15–23.17, you may need to create larger data sets to see a significant performance difference.

23.15 *(Summarizing the Words in a File)* Reimplement Fig. 17.22 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

23.16 *(Summarizing the Characters in a File)* Reimplement Exercise 17.10 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

23.17 *(Summarizing the File Types in a Directory)* Reimplement Exercise 17.11 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

23.18 *(Parallelizing and Timing 60,000,000 Die Rolls)* In Fig. 17.24, we implemented a stream pipeline that rolled a die 60,000,000 times using values produced by `SecureRandom` method `ints`. Use the same timing techniques you used in Exercise 17.25 to time the original stream pipeline's operation, then perform and time the operation using a parallel stream. Any improvement?

23.19 *(Calculating the Sum of the Squares)* Section 17.7.3 used `map` and `sum` to calculate the sum of the squares of an `IntStream`'s values. Reimplement stream pipeline in Fig. 17.9 to replace `map` and `sum` with the following `reduce`, which receives a lambda that does *not* represent an associative operation:

```
.reduce((x, y) -> x + y * y)
```

Error-Prevention Tip 17.2 cautioned you that `reduce`'s argument *must* be an associative operation. Execute the reimplemented stream pipeline using a parallel stream. Does it produce the correct sum of the squares of the `IntStream`'s values?