The program **AnimationThreadDemo** demonstrates the use of threads in Java programs. The program starts up with the **AnimationThreadDemo** class **run** method running in its own thread. The **run** method creates a new thread of the class **ExpandingBall**, stores it in the variable **expandingBall**, and calls its **start()** method. The **expandingBall** thread begins to run and animates a ball repeatedly expanding from a small size to a large size inside of a box. While the **expandingBall** thread is running, the **main-run** method continues to run in its own thread. It waits for mouse clicks and processes each click by calling the **processClick** method of the **ExpandingBall** class on the **expandingBall** object. A click inside the ball causes the animation to stop (if running) and start (if not running). A click outside the ball, but within the smallest rectangle enclosing the ball, causes the animation to stop for good. In this laboratory session you will experiment with threads and synchronization by making several modifications to the **AnimationThreadDemo.java** program.

1. Modify the program to construct and monitor an arbitrary number of expanding ball animations to be run simultaneously in the window. The program should have a constant **N** that specifies the number of animation threads. It should construct **N** instances of the **ExpandingBall** class, store them in an array, and start each of them. It should process each mouse click by calling the **processClick** method on each animation thread instance. It should terminate when none of the animation threads are alive.

2. The program is inefficient in the following way. Whenever the user clicks to stop an animation, the animation thread does not actually stop running. Instead it loops continuously checking its **running** variable to see if it should be updating the animation image, or doing nothing. This sort of loop is called a “busy wait”. It is wasteful because it takes computational resources away from other threads that might need them. You should remove this inefficiency in the following way:

   a. Define a **Gate** class to have two methods:
      i. The synchronized method **public void waitAtGate()** simply calls **wait()** inside of a try block, catching an **InterruptedException**.
      ii. The synchronized method **public void notifyAllAtGate()** simply calls **notifyAll()**.
   b. Arrange that the **ExpandingBall** constructor creates an instance of the class **Gate**, and stores it in an instance variable called “gate”.
   c. Modify the **run** method of **ExpandingBall** so that it calls **gate.waitAtGate()** whenever it notices that the animation should stop (i.e., the **running** variable is **false**).
   d. Modify the **processClick** method of **ExpandingBall** so that it calls **gate.notifyAllAtGate()** to bring the thread out of the waiting state and resume its execution.

In this implementation, each **Gate** object allows the **AnimationThreadDemo** class **run** thread to interact with one particular **ExpandingBall** thread. The **ExpandingBall** thread waits at its **gate**, by calling **gate.waitAtGate()**, whenever it notices that **running** is **false**. The **AnimationThreadDemo** class **run** thread brings the **ExpandingBall** thread out of the waiting state by calling **gate.notifyAllAtGate()** whenever a mouse click indicates that the animation should begin once again, or when the animation should be terminated forever.