A point previously made about threads, `notify`, and `notifyAll` must be stressed. The priority of a thread does not determine whether it is notified (in the case of using the `notify` method) or in what order multiple threads are notified (in the case of using the `notifyAll` method). Therefore, you should never make assumptions about the order in which threads are awakened as a result of calls to these methods. In addition, you should never make assumptions about the scheduling of a thread during preemption. Thread scheduling is implementation-dependent and varies by platform. It is unwise to make this type of assumption if your code is to be portable.

PRAXIS 54: Use spin locks for `wait` and `notifyAll`

When synchronizing multiple threads that access common data, you often need to use a notification mechanism with the `wait` and `notifyAll` methods. Using these methods requires the application of the spin-lock pattern so that your code works properly.

To illustrate, consider code that implements a robot controller. The `RobotController` class controls the actions of various robots connected to a computer. The controller and robots all run on separate threads. The controller provides a table of commands that is stored in a `static commands` field of the `Robot` class. Any robot can execute the commands. When a robot is finished executing the commands, it sets the `commands` variable to `null` and waits for more commands from the controller.

The robots wait for commands by a call to the `wait` method on the controller object. When the data arrives, they perform the commands in the table and then return to the wait state. The code for the controller and two robots might look like this:

```
class Robot extends Thread
{
  private static byte[] commands;
  private RobotController controller;
  public Robot(RobotController c)
  {
    controller = c;
  }
  public static void storeCommands(byte[] b)
  {
    commands = b;
  }
```