```
            notifyAll();              //Notify all threads.
        }
    }
```

Notice that the `run` method of the `Robot` class first acquires the lock for the `RobotController` object at //1. This is done so the `Robot` objects can operate on the table of commands without the `RobotController` object changing it. Note that the `loadCommands` method of the `RobotController` class, at //4, acquires the same object lock. Synchronizing this method ensures the robot command table is not altered when it is in use.

After the lock is acquired at //1, the `Robot` code then checks to see if there is a table of robot commands to process at //2. If such a table does not exist, the code executes a call to `wait` on the `RobotController` object at //3. This call releases the lock acquired at //1 and goes into a wait state. When the `loadCommands` method of the `RobotController` class is called, a `notifyAll` is issued, thereby waking up all of the `Robot` threads. To run, each thread must first reacquire the `RobotController` object lock. Only one thread acquires this lock at a time; the others must wait. The `Robot` thread that gets the lock can now access the table of commands and move the robot accordingly.

This code has a bug that causes it to fail in certain situations. Consider the following sequence of events:

1. An object of the `RobotController` class is created along with two objects of the `Robot` class. Each object runs on its own thread.
2. The first `Robot` thread checks to see whether the `commands` variable is `null` at //2.
3. The `commands` variable is `null`, so the first `Robot` thread blocks at //3 with a call to `wait`.
4. The second `Robot` thread checks to see whether the `commands` variable is `null` at //2.
5. The `commands` variable is `null`, so the second `Robot` thread blocks at //3 with a call to `wait`.
6. The `loadCommands` method of the `RobotController` class is called with a table of commands.