Chapter 6

Programming for the leJOS Environment

Solutions in this chapter:

- **Designing Java Programs to Run in leJOS**
- **An Advanced Programming Example Using leJOS**
- **Debugging leJOS Programs**
- **Testing leJOS Programs**

☑ **Summary**

☑ **Solutions Fast Track**

☑ **Frequently Asked Questions**

Introduction

In the previous chapter, we laid the groundwork for programming the RCX using Java and the leJOS environment. This chapter will contain a more advanced programming example. We have chosen the classic line-following challenge because it is well known in the LEGO MINDSTORMS community, it is good for demonstrating program design, and it's very easy to understand. The challenge is to make a robot follow a black line drawn on white paper. The RIS comes with such a paper test course, suited exactly for this challenge.

Some of the pitfalls to watch for include creating objects which go out of scope, thereby trashing memory and tying up the event listener thread for too long, causing missed sensor readings. These pitfalls are related to leJOS in general. We'll also address specific problems concerning the line following example, like how to get the robot back to the line if it's diverted, or how to prevent the robot from leaving the line in the first place. We'll also explore a design paradigm for robotic control called *subsumption architecture* in an extension to the original line-following code. This architecture was originally developed by R.A. Brooks in 1986, and is built upon layering and prioritizing different lower behaviors to achieve a more complex higher order behavior. Further information can be found at http://ai.eecs.umich.edu/cogarch0/subsump.

For the design of your software, it's best to create a good object-oriented (OO) design, using your preferred notation (UML happens to be mine) to achieve the easiest maintainable and extendable code, once implemented. The subsumption architecture will allow you to focus on perfecting individual behaviors of the robot independently, thus allowing you to test a partly implemented system and then expand it by adding additional behavioral code all in a modular fashion.

Designing Java Programs to Run in leJOS

Since Java is an object-oriented programming platform, object-oriented design techniques apply to programs for Java on the RCX as well. But I encourage you to never forget while doing the program design that this is Java for a *memory-limited embedded device*. In my opinion, it is crucial, right from the design phase of the program, to understand and take into consideration the conditions under which a program will run. The old way of doing OO design taught people to "just do" OO design. It did not matter how the finished system would be run, nor which programming language should be used since these were implementation details.

Well, I can tell you that it does matter since it is excruciatingly difficult to follow an OO design using COBOL as your programming language. Also, some OO design patterns do not take into account that when implemented, many objects will need to be created. This, of course, is fine when your program is run on a mainframe, but you really need to bend the design in unfavorable ways to cram it into an RCX, for instance.

So, what is it we really need to take into account when designing a Java program for the RCX? Memory, memory, and more memory!

Using Memory Wisely

How can we best use the available RCX memory for our programs? Well, we have to remember two things. First, we do not have a lot of it to begin with, and second, if we allocate some of it, we cannot return it to the leJOS system to use it again. This basically boils down to a few guidelines (or warning signs) to look for in the code:

- Only allocate objects you really need.
- Design your objects to be mutable; this allows you to reuse them.
- Do not allocate objects inside loops (this is not always bad if they are really needed).
- Consider using *byte*s and *short*s instead of *int*s. They take up less memory and are usually quite adequate for holding sensor readings and so on.
- Use the main thread. Do not just park it.

With that said, try not to compromise on writing nicely OO separated code. It is possible to have a neat OO-designed Java program given the restrictions imposed on you, but remember, you do not get extra points for having lots of memory free, especially when your program terminates because of an unmaintainable and overly complex code. Of course, should you run into "memory trouble," you have to know what to look for, and how to improve the code. Having your program do what it is supposed to, with a clean design that happens to use up all the memory is far better than having sloppy code that has chunks of free memory it doesn't use.

If you are puzzled by the enormous amount of memory your program consumes, try looking at the implementation of some of the base classes you use. The code is not that hard to understand, as we'll demonstrate in the next section with *StringBuffer*.

Using the Right Java Classes (and Using Them Correctly)

A good way of demonstrating memory optimization for a number of allocated objects is the optimization of allocating instances of *java.lang.String*. The reason for this is that a *String* is immutable, which means that once a *String* is constructed, the characters it uses of can never be changed. Therefore, you have to create a new *String* object if you wish to represent another string. The standard Java solution for this is to use an instance of *StringBuffer* to build your dynamic string content and then turn that into a *String* (using *StringBuffer*s *toString* method) for printing purposes and so on. This technique also applies to leJOS. Consider the program shown in Figure 6.1 (this can be found on the CD that accompanies this book, in the /string directory).

Figure 6.1 StringTest Shows the Usual Way of Manipulating String Objects (StringTest.java)

```
import josx.platform.rcx.TextLCD;
import josx.platform.rcx.Button;
import josx.platform.rcx.LCD;

public class StringTest {
    public static void main (String[] args) throws InterruptedException
    {
        String ha = "HA";
        TextLCD.print (ha);
        Button.VIEW.waitForPressAndRelease ();

        ha = ha + ' ' + ha;
        TextLCD.print (ha);
        Button.VIEW.waitForPressAndRelease ();

        LCD.showNumber ((int)Runtime.getRuntime().freeMemory());
        Button.VIEW.waitForPressAndRelease ();
    }
}
```

Well, it looks innocent enough, but actually there is no explicit use of the new statement, so at first glance it doesn't appear as if any objects are allocated. In truth, though, a lot of objects are created. First, the initial assignment of *ha* creates a *String* object, then in the line *ha = ha + ' ' + ha*; a *StringBuffer* is allocated using its default constructor, and 3 *append* methods on it are called. That does not sound frightening, but the leJOS version of the *StringBuffer*s default constructor allocates an internal *char* array of size 0, and upon each *append* call, this buffer is replaced with a new one if it does not have room for the characters you are about to append. To make matters worse, *String* (to preserve its immutability) allocates another *char* array when told to return itself as one using the *toCharArray()* method (which *StringBuffer* will call to append the *String*). *StringBuffer* also converts an appended *char* to first a *char[]* of size one, then a *String*, and then calls *append* with that *String*, which we just saw will make *String* create another *char* array. Finally, a *String* is created using the *StringBuffer*s *toString* method (whew!).

In total, that innocent-looking line allocates one *StringBuffer*, two *String*'s, and 9 *char[]*, for a total of 12 new objects, of which only two can actually be (re)used—the resulting *String* and the *StringBuffer*. As you can see, we finished the program by outputting the amount of available memory. The number says 2348!

So, let's check out the version shown in Figure 6.2 (also found on the CD), which is the standard Java solution to this problem, using a *StringBuffer* directly.

Figure 6.2 StringBuffer Shows the Usual Java Optimization for StringTest (StringBufferTest)

```
import josx.platform.rcx.TextLCD;
import josx.platform.rcx.Button;
import josx.platform.rcx.LCD;

public class StringBufferTest {
    public static void main (String[] args) throws InterruptedException
    {
        String ha = "HA";
        TextLCD.print (ha);
        Button.VIEW.waitForPressAndRelease ();

        StringBuffer bf = new StringBuffer (5);
        bf.append (ha);
        bf.append (' ');
```

Continued

Figure 6.2 Continued

```
        bf.append (ha);
        TextLCD.print (bf.toString());
        Button.VIEW.waitForPressAndRelease ();

        LCD.showNumber ((int)Runtime.getRuntime().freeMemory());
        Button.VIEW.waitForPressAndRelease ();
    }
}
```

Not much has actually happened. we have specified a size for the *StringBuffer*, which will eliminate the *char[]* allocated inside *StringBuffer* when it determines it doesn't have enough space allocated for the append operation to succeed, and that's all. The amount of memory available here hasn't improved dramatically—it's only up to 2364. A fast fix is to notice that TextLCD will actually print a *char[]* as well as a *String*, so changing the line *TextLCD.print (bf.toString();* into *TextLCD.print (bf.getChars());* will eliminate one *String* creation (and that *String*'s internal creation of a *char[]*) which will give you 2400 free bytes.

We still do not have a lot of free memory, and we are not really doing anything spectacular. The next move is to realize that *StringBuffer* itself is a fairly large class, so let us get rid of it entirely—Figure 6.3 (also found on the CD) shows a version which does that using *char[]* directly.

Figure 6.3 CharTest Shows the Ultimate Optimization (CharTest.java)

```
import josx.platform.rcx.TextLCD;
import josx.platform.rcx.Button;
import josx.platform.rcx.LCD;

public class CharTest {
    public static void main (String[] args) throws InterruptedException
    {
        char[] ha = "HA".toCharArray();
        TextLCD.print (ha);
        Button.VIEW.waitForPressAndRelease ();

        char[] bf = new char[5];
        byte curpos = 0;
```

Continued

Figure 6.3 Continued

```
        for (byte i = 0; i < ha.length; i++) {
          bf[curpos++] = ha[i];
        }
        bf[curpos++] = ' ';
        for (byte i = 0; i < ha.length; i++) {
          bf[curpos++] = ha[i];
        }
        TextLCD.print (bf);
        Button.VIEW.waitForPressAndRelease ();

        LCD.showNumber ((int)Runtime.getRuntime().freeMemory());
        Button.VIEW.waitForPressAndRelease ();
    }
}
```

How much did this version improve matters? Well, by getting rid of *StringBuffer* we have boosted the amount of free memory to 5774 bytes—rather good, don't you think?

Could it be improved even further? Yes, but not much. You can get rid of the initial *"HA"* string and place the individual characters in a *char* array directly. Thus, the initialization of variable *ha* will look like:

```
char[] ha = new char[2];
ha[0] = 'H';
ha[1] = 'A';
```

This will save you an additional 14 bytes, so it is hardly worth the trouble, unless you are really making large programs or collecting large amounts of data.

The lesson hopefully learned is that things are not always as they seem. The first version is definitely the least complex to understand, but it burns memory like crazy. And to understand fully what is going on, you have to look at the implementation of the used base classes. In this case, knowing the internal behavior of *String* and *StringBuffer* allows you to come up with a solution to the problem. But keep in mind that using *TextLCD* itself, of course, has a cost (as mentioned in the previous chapter), which amounts to around 1500 bytes. So, unless you really need the text output (it can often improve the usability of programs) use *LCD* instead.

By the way, what we've looked at is not leJOS-specific since the previous examples behave exactly the same in standard Java. The only difference is that in leJOS the space allocated is never reclaimed. Therefore, you pay more dearly for the unneeded memory allocations (and also the default size of a *StringBuffer* is 16 characters in standard Java, instead of 0 in leJOS).

An Advanced Programming Example Using leJOS

As mentioned earlier, we will show you how to program the classic line-following robot in leJOS—see the robot shown in Figure 6.4. This robot has two rear wheels powered by a single motor, and is steered by its one front wheel, which is turned by another motor. The downward-looking light sensor is placed in front of the steering wheel and turns with it. Because of the sensor's position, even small adjustments in steering will move the sensor a great deal. This will make the sensor contact the line's edge more quickly, and thus greatly reduce the chance of the robot steering away from the line. The robot uses only this single sensor and it can readily be built using a standard LEGO Robotic Invention System kit.

Figure 6.4 The Line-following Robot

The prerequisite of the line-following challenge is to stick to the left side of the line. At the edge, the light sensor will return a value between what is considered black and what is considered white. Initially, the robot will be placed on white, to calibrate the white value, and then placed on black for the same purpose. Afterward, when you place it on the line's edge, it will start following it. Path correction will involve turning right if the sensor returns a value considered "white," meaning we have strayed off the line to the left, and similarly turning left if the sensor reports "black," meaning the robot has now turned into the line. The driving motors will be set to full power so the robot follows the line as quickly as possible.

Let's start by making a nice OO design for it. From reading the preceding description, you can see you need some code for calibration, some for reading the light sensor, some for driving forward, and some for steering based on the value read by the sensor.

We have chosen to use the *TextLCD* for giving directions to the user under the initial calibration phase. Figure 6.5 is a UML class diagram showing the classes needed.

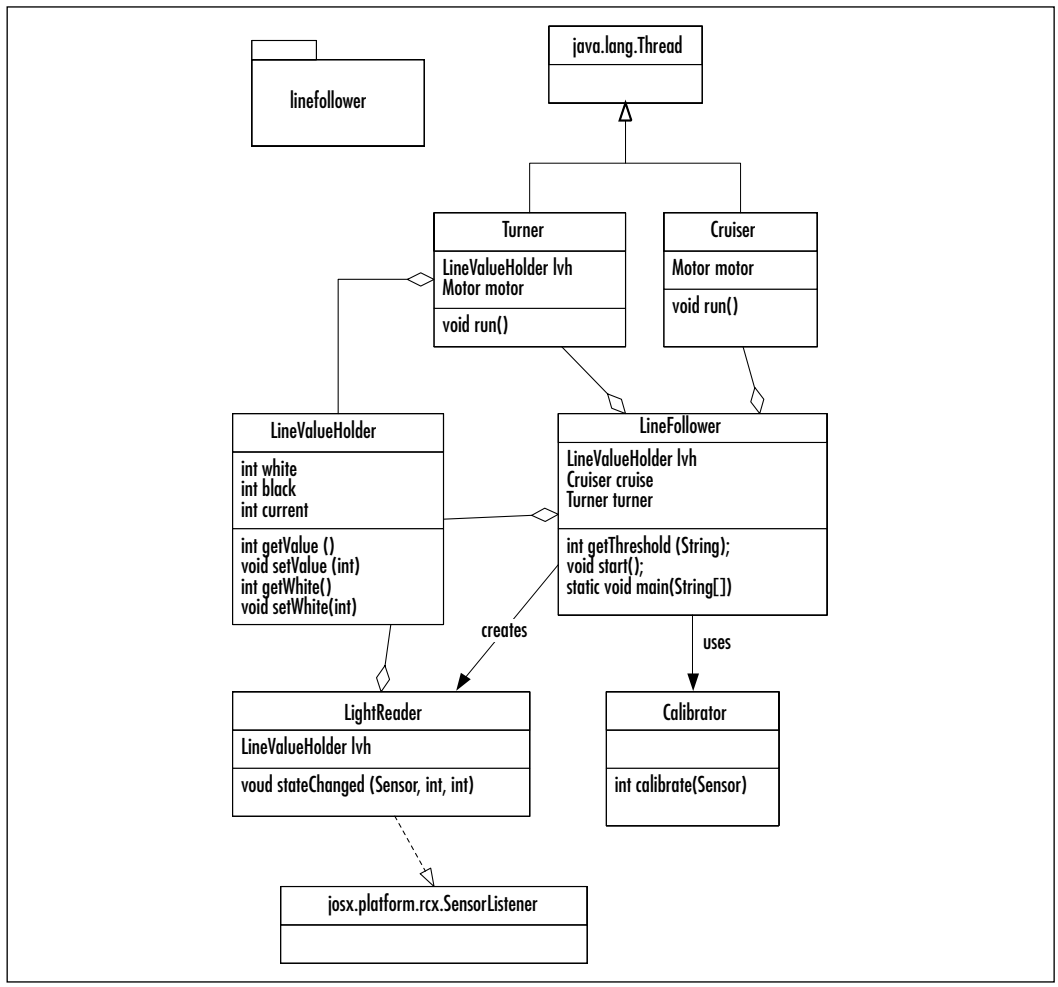
As you can see, a heavily used class is the *LineValueHolder*, but actually only one instance of it will be created. Objects aggregated from it will share the same instance.

The *LineValueHolder* is an example of a mutable object. It is used to synchronize access to the values supplied from the light-sensor (it is actually an implementation of a solution to the classical readers-writers problem). Here one class (the *LightReader*) will act as the only writer; it will be called when the light-sensor has changed its value, then it will update the current sensor reading held by the *LineValueHolder* by calling *setValue*, which will wake up any waiting readers.

The reader (in this case an instance of *Turner*) will continuously call *LineValueHolder*'s *getValue()*. This call will block until later notified by the *LightReader*'s call to *setValue,* signaling that a new value has become available. Notice that this code does not hold any queue, so it is possible that readers might miss some values, but for this application that probably does not matter as the light sensor tends to change values often. The *LineValueHolder* also holds the program's interpretation of white and black; those values are placed into it once at the start of the program using the *Calibrator*.

What about the user interaction? For simplicity's sake, we have chosen the not-so-pretty approach—that is, to place the user interface directly in the main class *LineFollower*'s constructor; the prettiest OO design would probably be to place the user interface and user interaction code into a separate UI class.

Figure 6.5 The UML Class Diagram for the Line-follower



Finally, the two *Thread* subclasses, *Turner* and *Cruiser*, are responsible for movement of the robot, and as the name implies, *Turner* turns the robot left or right according to the current sensor reading, and *Cruiser* is responsible for driving forward as fast as possible.

Configuration of *Sensor.S1*, which the light-sensor is attached to, is done in the *LineFollower*'s constructor.

Now, let's take a look at the implementation of those classes one by one. First, the important *LineValueHolder*. (The code examples in Figures 6.6 through 6.11 are provided on the accompanying CD of this book in the /linefollower/original/linefollower directory for Chapter 6.)

Figure 6.6 LineValueHolder Contains Main Synchronization Between Sensor Reading Code and Steering Code (LineValueHolder.java)

```
package linefollower;

public class LineValueHolder {
  public int white;
  public int black;
  public int current;
  public LineValueHolder() {
  }
  public synchronized void setValue (int value) {
    current = value;
    this.notifyAll(); //notify all waiting readers
  }
  public synchronized int getValue () throws InterruptedException {
    this.wait(); //wait until a new value is read
    return current;
  }
  public void setWhite (int value) {
    white = value;
  }
  public int getWhite () {
    return white;
  }
  public void setBlack (int value) {
    black = value;
  }
  public int getBlack () {
    return black;
  }
}
```

The most important and interesting methods are the two synchronized methods, *getValue* and *setValue*.

The *setValue* method's most important feature is that it is very short, and thus executes very fast. It will be called by the *LightReader* within its *stateChanged*

method, which in turn is called by the leJOS sensor reading thread. So, the longer you tie up that sensor reading thread, the more sensor readings you will miss.

The two methods work in parallel, as calls to *getValue* will block until another call has been made to *setValue*. The reason this will work is that, as mentioned previously, sensor-readings will change! Let's now take a look at our *SensorListener*, which calls *setValue* (see Figure 6.7).

Figure 6.7 LightReader Is Responsible for Delivering Sensor Readings to the LineValueHolder (LightReader.java)

```
package linefollower;
import josx.platform.rcx.SensorListener;
import josx.platform.rcx.Sensor;

public class LightReader implements SensorListener{
  LineValueHolder lvh;

  public LightReader(LineValueHolder lvh)
  {
    this.lvh = lvh;
  }
  public void stateChanged (Sensor sensor, int new_value, int old_value)
  {
    lvh.setValue (new_value);
  }
}
```

As you can see, the *stateChanged* method is very short, so it does not tie up the sensor reading thread as just mentioned. It simply calls the *LineValueHolder*'s *setValue* method (which, as you saw previously, was very short as well). This call transfers the new value as read by the *Sensor*.

Let's move on to the Cruiser, shown in Figure 6.8.

Figure 6.8 Cruiser Is Doing the Driving (Cruiser.java)

```
package linefollower;
import josx.platform.rcx.Motor;

public class Cruiser extends Thread {
```

Continued

Figure 6.8 Continued

```
  Motor motor;

  public Cruiser(Motor motor) {
    this.motor = motor;
    motor.setPower (7);
  }
  public void run () {
     motor.forward();
  }
}
```

As you can see, it simply sets the power level to maximum and then when it is started, sets the motor going forward.

The Turner thread (see Figure 6.9) is a bit more complicated, as it behaves differently based upon the light sensed.

Figure 6.9 Turner Is Responsible for Steering the Robot According to the Sensor Readings (Turner.java)

```
package linefollower;

import josx.platform.rcx.Motor;
import josx.platform.rcx.LCD;

public class Turner extends Thread {
  static final int HYSTERESIS = 4;
  Motor motor;
  LineValueHolder lvh;

  public Turner(Motor motor, LineValueHolder lvh) {
     this.lvh = lvh;
     this.motor = motor;
     motor.setPower (7);
  }
  public void run () {
     while (true) {
```

Continued

Figure 6.9 Continued

```
        try {
          int light = lvh.getValue();
   //show the current reading, great for debugging
          LCD.showNumber (light);
          if (light < lvh.getBlack() + HYSTERESIS) {
            motor.forward();
          } else if (light > lvh.getWhite() - HYSTERESIS) {
            motor.backward();
          } else {
            motor.stop();
          }
        } catch (InterruptedException ie) {
          //ignore
        }
      }
  }
}
```

The logic in the *run()* method, which is implementing what is interesting, is pretty simple. It compares the light reading with what is considered black—if it is less, it spins the motor forward. With the right wiring, this will turn the wheel, and thereby the robot, left. Similarly, if it is greater than what is considered to be white, it spins the motor backward, which should turn the robot right. Otherwise, we do not need to turn so we stop the motor.

NOTE

Hysteresis actually means "lag of effect," a term that comes from physics, where it means a delay in the observed effect when forces on a body change. This term has crept into software lingo with the meaning that you need to take this lag of effect in the physical world into account when programming. This is often done (as in the example in Figure 6.9) by adding/deducting a value in a comparison. This will also allow a program to be more immune to noise in sensor readings.

Can that ever work? Of course not. Remember, white is probably the highest value we will see, and black the lowest. So we simply deduct/add a constant called *HYSTERESIS* before the comparison, which will make an interval of values considered too white or too black which we need to react to. Notice that it is rather small (only four), but for our lighting conditions this value seems to work. Plus, it's a place where you can tune your behavior.

The last of the helper classes is the *Calibrator*. Its *calibrate* method simply takes the average over 20 successive light-sensor readings, as shown in Figure 6.10.

Figure 6.10 Calibrator Is Used to Define the Interpretation of Black and White (Calibrator.java)

```
package linefollower;
import josx.platform.rcx.Sensor;

public class Calibrator {
  private static final int  NUMBER_OF_SAMPLES = 20;
  Sensor sensor;

  public Calibrator(Sensor sensor) {
     this.sensor = sensor;
  }

  public int calibrate () {
       int sum = 0;
       for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
         sum += sensor.readValue ();
       }
       return sum / NUMBER_OF_SAMPLES;
  }
}
```

Now, look to Figure 6.11 for the main class, *LineFollower*, whose purpose is to set things up, drive the user interaction, and finally kickoff the controlling threads.

Figure 6.11 The Main Program Class Does Mainly Setup but also Drives the User Interface (LineFollower.java)

```
package linefollower;

import josx.platform.rcx.LCD;
import josx.platform.rcx.TextLCD;
import josx.platform.rcx.Sound;
import josx.platform.rcx.Motor;
import josx.platform.rcx.Sensor;
import josx.platform.rcx.SensorConstants;
import josx.platform.rcx.Button;

public class LineFollower {

  LineValueHolder lvh = new LineValueHolder();
  Cruiser cruise;
  Turner turner;

  public LineFollower() throws InterruptedException {
    Sensor.S1.setTypeAndMode (SensorConstants.SENSOR_TYPE_LIGHT,
                              SensorConstants.SENSOR_MODE_PCT);
    Sensor.S1.activate();

    waitForUser ("white");
    lvh.setWhite (getThreshold());
    waitForUser (null);

    waitForUser ("black");
    lvh.setBlack (getThreshold());
    waitForUser (null);

    Sensor.S1.addSensorListener(new LightReader(lvh));

    cruise = new Cruiser (Motor.B);
    turner = new Turner (Motor.A, lvh);
  }
```

Continued

Figure 6.11 Continued

```
  public void waitForUser (String message) throws InterruptedException
  {
    if (message != null) {
      TextLCD.print (message);
    }
    Sound.twoBeeps ();
    Button.VIEW.waitForPressAndRelease();
  }

  public int getThreshold () {
    Calibrator calib = new Calibrator (Sensor.S1);
    int value = calib.calibrate ();
    //show calibration value, good for tuning the HYSTERESIS constant
    //in the Turner class
    LCD.showNumber(value);
    return value;
  }

  public void start ()
  {
     //start threads
     cruise.start();
     turner.run();
  }

  public static void main(String[] args) throws InterruptedException {
    LineFollower lineFollower = new LineFollower();
    lineFollower.start ();
  }
}
```

The code for *LineFollower* is fairly simple. A *LineFollower* object is created in the *main* method. Within the constructor, the *Sensor.S1* is then configured for light readings. Then the user is prompted with the text "white" which should serve as an instruction to place the robot's light-sensor over the white surface. When done, the user must press the **View** button for white calibration to begin,

the calibration value is then displayed, and the user is again expected to press the **View** button to continue. This process resumes, but this time with the text "black" as prompt. When the user presses the **View** button at the end, the program continues its execution, so the user must place the robot near the left edge of the line, before doing the final **View** button press. Notice that we have also used a double beep to attract the user's attention, indicating something is needed from her. The rest of the program consists of only two things—first, attaching the *LightReader* as a listener to *Sensor.S1*, and the construction and start of the *Turner* and *Cruiser* threads. If all goes well, your robot will start following the black line.

Now, it was promised that some "mistakes" would be purposely included in the program to make it use more memory than actually needed. We will continue to refrain from correcting them (as long as we don't run out of memory), because they increase the readability and usability of the program. The memory optimized version can also be found on the accompanying CD in the /linefollower/optimized directory.

We will now go over the "mistakes" and at the same time explain how to remove them and thereby optimize the program.

- Notice in Figure 6.11 that the method, *getThreshold*, is called twice in the *LineFollower*'s constructor, and that within it a *Calibrator* is instantiated. That means two *Calibrator* instances and one *Calibrator* that does pure calculations. In fact, its sole method is actually thread-safe since it only works on method local variables, so a fast optimization is to simply make it into a static method and not allocate any *Calibrator* objects at all. If the *calibrate* method had not been thread-safe, we could still have done that because in our case, the calls to *calibrate* occur from only one thread, the main thread of the program.
- In the *LineValueHolder* (Figure 6.6) *int*s are the used type for the instance variables holding the sensor readings. But as light-sensor readings are percentages their values lies between 0 and 100, and a Java byte which has the range [-128:127] is quite adequate for holding those readings. So memory can be saved by using *byte*s instead of *int*s.
- It is really not so nice that three references to the *LineValueHolder* instance need to be kept at various places. The one in the *LineFollower* is easy to get rid of. Just move the construction inside the *constructor* and lose that *instance* variable. But to get rid of all three of them, we will instead apply the Singleton pattern, as identified in the book *Design*

Patterns by the Gang Of Four (Gamma, Helm, Johnson, and Vlissides). So, the changes to the *LineValueHolder* construction will look like:

```
...
private static LineValueHolder instance = null;
  public static LineValueHolder getInstance () {
    if (instance == null) {
      instance = new LineValueHolder ();
    }
    return instance;
  }

  private LineValueHolder() {
  }
...
```

- Now, take a look at the *Cruiser* thread (Figure 6.8). Actually the *run* method just starts the motor running and then terminates, thereby terminating the *Thread*. So it will be okay to just call the method directly from the *LineFollower*'s *start* method (using the main thread). To improve things further, you can even remove the *Thread* extension, thereby making the constructed object smaller.
- Possible removal of a *Thread* extension also holds for the *Turner* thread because we used the already available main thread to call its *run* method directly, instead of indirectly through a *Thread.start()* call.
- *Cruiser* does not actually need to be instantiated at all, and can thus have its methods made static. We have not made that change, as it will make the use of *Cruiser* vastly different from that of the *Turner*.
- Finally, take another look at the *LineFollower* in Figure 6.11. All the methods are executed by the main thread, so no synchronization is needed. Thus, we do not really need to create any *LineFollower* objects. We can do that by turning *LineFollower*'s instance methods into static equivalents. The constructor we will change into a static *init* method, which gets called by the *main* method.

We ran a slightly modified version of the original program with the one change that instead of continuously outputting the light reading, it outputted the amount of free memory. Then we did the same with the modifications just listed.

The result was that for the original, 3684 bytes were free, and for the modified version, 3814 bytes were free. Even though not much memory was saved, we haven't sacrificed any readability, usability, or maintainability of the program. Notice that in the optimized version on the CD we have chosen to keep the use of *TextLCD*, as opposed to using *LCD*, since it improves usability of the program.

Here is a list of other improvements you can try and implement yourself.

- As it is now, the program is stopped by pressing the **On-Off** button, which turns off the RCX completely. A nice feature would be a way to stop the program, perhaps by pressing the **Run** button; try adding a *ButtonListener* to do that. Its action could probably be to call *interrupt* on the main thread, to force it to break out of the endless loop in *Turner*'s run method. Notice that this demands a change in *Turner* as well, as it actually catches *InterruptedException*, ignores it, and continues with its business. And, of course, you need to keep a reference to the main thread somewhere (get that reference using the *Thread.currentThread()* method).
- The calibration routine could be improved to take the average of light readings made at different places along the course. This change would need some directions to the user to move the RCX between subsequent readings.

Designing & Planning…

Synchronization on Static Methods

Remember that no instances of *java.lang.Class* exist in leJOS, so you cannot synchronize on static methods. This is the reason you need to analyze your multithreaded programs. Make sure that when using static methods, no two threads will execute them simultaneously, or make sure the methods are thread-safe.

Controlling the Steering

If the robot, going too fast, passes entirely over the line to the right side of the black track, it will likely go into a never-ending spin around itself. This section will try to come up with two solutions for that.

The first solution will try to solve the problem by restricting the amount of turning allowed by the steering wheel, making it less likely that the robot will pass over the line.

The second solution is to try to find a way back to the left side of the line. This is achieved by determining that the line has been passed, and then overruling the robot's standard line-following behavior with a higher prioritized behavior which will hopefully bring the robot back on track; this solution introduces the subsumption architecture.

Before you attempt to control the steering, you will need to acquire some feedback of where the steering front wheel is positioned at any given time. To do this, we need to add a longer steering axle to the original design of the robot. This change allows us to put a rotation sensor on top of it, which is then connected to input 2 on the RCX. As a result, this rotation sensor gives us a way to measure the position of the wheel.

Restricted Steering

The first thing we'll do is make a programming change that controls how far to one side the wheel is allowed to turn. This, of course, will prevent the robot from making very sharp turns, but at least for the LEGO test pad we'll use here, no really sharp turns exist. The idea behind this is that, if the RCX doesn't do any sharp turns, it will not pass entirely over the line, and the spin-forever effect will go away.

In the program, we'll simply change *Turner* into a version utilizing not only a *Motor* but also a *Sensor,* or more specifically, a *Sensor* configured for rotational readings. To ease the programming, let's use the *Servo* class, which is precisely suited for this kind of control. The *Servo* combines a *Motor* and a *Sensor* so you can tell the *Motor* to turn to a certain axis point. It is important to notice that, as the servo keeps a number representing the absolute position of the *Motor*, you will need to start the robot with the steering wheel in a neutral position so it is going straight. The change is then to have a maximum turn of two to either side, a value determined from previous experiments. A lot of gearing exists between the rotation sensor and the motor, so that is the easiest way to find it. This solution actually solves the problem in a very simple manner. At least on this test course (in my experiences), the spinning behavior did not occur after the change was made. Prior to that, it was very frequent. In fact, usually two consecutive laps of the course could not be made before the spinning occurred.

Figure 6.12 shows the modified *Turner* code (for simplicity, we have chosen to refrain from my traditional approach where the *Motor*s and *Sensor*s used are determined in the main class and distributed to the objects using them, as parameters;

instead, this *Turner* version explicitly uses *Sensor.S2*). (You can locate a complete version of the line follower with this modification on the CD, in the /linefollower/steering1/linefollower directory.)

Figure 6.12 A Limited Turn Turner (Turner.java)

```
package linefollower;

import josx.platform.rcx.Motor;
import josx.platform.rcx.LCD;
import josx.platform.rcx.Servo;
import josx.platform.rcx.Sensor;

public class Turner extends Thread {

  static final int HYSTERESIS = 4;
  Servo servo;

  public Turner(Motor motor) {
     this.servo = new Servo (Sensor.S2, motor);
  }

  public void run () {
     LineValueHolder lvh = LineValueHolder.getInstance();
     while (true) {
        try {
          int light = lvh.getValue();
          LCD.showNumber (light);
          if (light < lvh.getBlack() + HYSTERESIS) {
            servo.rotateTo (-2);
          } else if (light > lvh.getWhite() - HYSTERESIS) {
            servo.rotateTo (2);
          } else {
            servo.rotateTo(0);
          }
        } catch (InterruptedException ie) {
          //ignore
```

Continued

Figure 6.12 Continued

```
            }
        }
    }

}
```

Getting Back to the Line

The second solution to the same "never-ending turn" problem is a much more advanced approach, involving the control concept called *subsumption architecture*. Also, the solutions tactic is a different one. It does not try to get the robot to avoid passing the line. Instead, it tries to determine when it has done so, and then runs some code in hopes of returning the robot to the left side of the line.

The idea of *subsumption architecture* is to first divide our program into subtasks or behaviors. Each subtask is responsible for controlling one specific behavior of our robot. This behavior can be simple, like backing up when a touch sensor is hit, or it can be complex, like our entire follow-the-line behavior. The next thing we need to do is prioritize the behaviors. The implementation is then made in a way so that only the highest prioritized behavior is actually running, this makes it possible to have multiple behaviors competing for the same resources (motors, sensors, display, and so on) to coexist. It is, of course, crucial that the individual behaviors release their control when they do not need it, to allow lower prioritized behaviors a chance to run.

Designing & Planning…

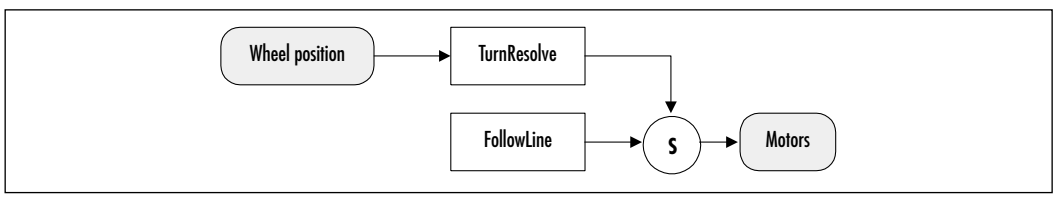
Line-following Software Design for a Differential Drive-based Robot

Had the robot been built using a differential drive as used in the previous chapter, the steering and driving straight behaviors would have been competing for the control of the motors, since this drive platform uses the same set of wheels for driving and steering. So, in this case, it would make perfect sense to use a subsumption-based architecture for programming the line following in the first place.

If our line-following program had been made using this architecture, the turning behavior would be a higher priority behavior than the drive forward behavior. However, these two behaviors in our example are totally independent of each other, and can therefore coexist without competing for the control of motors.

This example illustrates just one possible situation where this architecture shows its strength. Every time we have multiple behaviors, triggered by different situations (like sensor reading) but in need of the same actuators to perform the needed action, this architecture is a good way to modularize our code and keep it maintainable. Subsumption architectures are usually designed using diagrams like the one shown in Figure 6.13, which indicates that the sensors (*Wheel position* in the diagram) triggers certain behaviors together. The diagram also shows the priority of the different behaviors in that higher priority behaviors are drawn above lower priority ones. The round shape with the S, for *subsume*, shows that the higher priority behavior can take the control of a resource (here *Motors*) away from a lower priority one. This kind of diagram does not say anything about how each individual behavior is to be designed. They can then be made using standard OO techniques. The subsumption architecture can be viewed as a way to make objects cooperate. If you are familiar with design patterns, think of this architectural concept as a pattern.

Figure 6.13 Subsumption Architecture for Our Solution to the Spin Effect Problem



As explained earlier, the diagram is read from top to bottom and left to right, so the highest priority behavior is the new *TurnResolve*, and is somehow triggered by a wheel position. The lowest priority behavior is our existing line-following behavior, which is not triggered by anything but just runs continuously when not subsumed by any higher priority behavior.

To incorporate this into our program, let's use a class that resides in the *josx.robotics* package, named *Arbitrator*. What it does is perform arbitration of prioritized behaviors (just what we need). Let's take a look at the *Arbitrator* and its related *Behavior* interface.

NOTE

In addition to the *Arbitrator* class and *Behavior* interface used in our example of subsumption architecture in use, the *josx.robotics package* contains other classes useful for programming navigating robots. These classes come in two types. The first bases navigation on timing constraints (it takes *n* seconds to turn 90 degrees) and the other bases navigation on readings from rotation sensors. They are provided with a common interface, so you can easily switch between the two implementations.

Arbitrator

The *Arbitrator* class defines only one method, called *start*, which takes no arguments, so starting the arbitration is very easy. Before calling it, you need to construct an *Arbitrator* object, and its constructor takes an array of objects implementing the *Behavior* interface. These objects must be preordered in an array in such a way that the *Behavior* at index 0 corresponds to the desired behavior with the lowest priority, and so forth. Note that the *start* method does not start a separate thread, and thus never returns.

NOTE

In some designs, some behaviors actually have equal priority, which is not a problem as long as they are not competing for the same resources. This situation cannot be handled by the *Arbitrator* class, because its implementation enforces a strict hierarchy of priorities.

Behavior

The *Behavior* interface must be implemented by classes being arbitrated by the *Arbitrator* class. These implementations, in turn, are responsible for implementing the desired behavior for a given priority. The methods you must implement include:

- **public void action ()** This method must implement the task that is to be performed when this *Behavior* is in control. Notice that the method must not go into a never-ending loop, which will cause the arbitration process to no longer work.

- **public boolean takeControl ()** The implementation must return *true* if this Behavior wishes to take control. Whether it will actually take the control, of course, depends on its position in the priority hierarchy.
- **public void suppress ()** Your implementation of this method will stop the current running action (stopping motors and so on). The method should not return until the running action has terminated (but it must return eventually for the arbitration process to work).

Resolving the Never-ending Turn

We will use the same mechanical design as in the previous solution (see the "Restricted Steering" section), a rotation sensor attached to the top of a longer steering wheel axle.

The strategy will be as follows: whenever we, during the line-following process, end up having turned all the way to the right, that must mean we have crossed the line and engaged in that devious death spin. This situation is indicated by the rotation sensor having a value of 5 (experimentally determined). The behavior we will be implementing to escape from the spin is realized in the *TurnResolve* class. It will subsume the line-following behavior when the rotation sensor shows 5, and then do something clever to get it back on the line.

First, we need to turn our existing line-following code into something implementing the *Behavior* interface, so we can add it to an *Arbitrator* object. The change simply wraps the *Turner* and *Cruiser* instances in a class *LineFollowBehavior* (see Figure 6.14) implementing *Behavior*. This is an application of the standard wrapper pattern, known from the GOF book. Additionally, the *Cruiser* and *Turner* have themselves been turned into separate *Behavior* implementations. This way they can perhaps be reused as stand-alone behaviors in another setting. To complete the picture, they can be studied in Figure 6.15 and 6.16 respectively. (The code for Figures 6.14 through 6.16 can be found on the CD in the /linefollower/steering2/linefollower directory.)

Figure 6.14 Making the Line-following Behavior Implementation (LineFollowBehavior.java)

```
package linefollower;

import josx.robotics.Behavior;

public class LineFollowBehavior implements Behavior
```

Continued

Figure 6.14 Continued

```
{

  Cruiser cruise;
  Turner turner;
  public LineFollowBehavior(Cruiser cruise, Turner turner) {
    this.cruise = cruise;
    this.turner = turner;
    turner.start();
  }

  public void action () {
     cruise.action ();
     turner.action ();
  }

  public boolean takeControl () {
    return true;
  }

  public void suppress () {
    cruise.suppress();
    turner.suppress();
  }
}
```

Figure 6.15 Cruiser as a Behavior Class (Cruiser.java)

```
package linefollower;
import josx.platform.rcx.Motor;
import josx.robotics.Behavior;

public class Cruiser implements Behavior{
  Motor motor;

  public Cruiser(Motor motor) {
```

Continued

Figure 6.15 Continued

```
    this.motor = motor;
  }

  public void action () {
     motor.setPower (7);
     motor.forward();
  }

  public boolean takeControl () {
    return true;
  }

  public void suppress () {
    motor.stop();
  }
}
```

As the main thread is now used to run the Arbitration process, the *LineFollowBehavior* will start the *Turner* instance as a separate thread (see Figure 6.16). This, in turn, will make the *takeControl* and *suppress* methods in *Turner* a little complicated. The idea is that a variable (*halted*) is used to signal if the steering is active. If it isn't, the thread is parked in a *wait* call. The Arbitrator's call to *action* will then wake up this thread again by calling *notifyAll*. Notice that both the *Turner* and *Cruiser* in their implementation of *takeControl* always return true, signaling they are always willing to run.

Figure 6.16 Turner as a Behavior Class (Turner.java)

```
package linefollower;
import josx.platform.rcx.Motor;
import josx.platform.rcx.LCD;
import josx.robotics.Behavior;

public class Turner extends Thread implements Behavior
{
  static final int HYSTERESIS = 4;
  Motor motor;
```

Continued

Figure 6.16 Continued

```
boolean halted = false;

public Turner(Motor motor) {
   this.motor = motor;
}

public synchronized void action () {
  motor.setPower (7);
  halted = false;
  notifyAll();
}

public boolean takeControl () {
   return true;
}

public synchronized void suppress () {
    motor.stop ();
    halted = true;
    notifyAll();
}

public void run () {
  LineValueHolder lvh = LineValueHolder.getInstance();
  while (true) {
    try {
      if (!halted) {
        int light = lvh.getValue();
        LCD.showNumber (light);
        if (!halted) {
          if (light < lvh.getBlack() + HYSTERESIS) {
            motor.forward();
          } else if (light > lvh.getWhite() - HYSTERESIS) {
            motor.backward();
          } else {
```

Continued

Figure 6.16 Continued

```
              motor.stop();
            }
          }
        } else {
          synchronized (this) {
            wait (); //block until notified
          }
        }
      } catch (InterruptedException ie) {
            //ignore
      }
    }
  }
}
```

Second, we must implement our high priority *TurnResolve* behavior. Note that because the behavior is defined by a *Sensor* that does not change value as opposed to one that does change value, we cannot implement this using a *SensorListener* but must explicitly read the *Sensor* value at some interval.

The "clever" thing to do to get back on the line is really not that clever: We first slow down, and straighten up the drive wheel (it's best to actually keep it a little turned, so we curve back towards the line). Then because the line is relatively thin compared to our robot, we should assume that the robot will be spinning on the white surface, and also be on the white surface right after it has straightened out its steering wheel. So we keep driving until we hit a black surface. This should be the right edge of the line (as the robot follows the left edge, it must have passed the line and now be on the right-hand side of the line). We then keep going until we hit white again. At this point, the robot is probably at an angle somewhat perpendicular to the line, so we must turn the steering wheel, and try to curve back until we once again hit black. This should then hopefully be on the desired left side of the line. You can see the implementation of *TurnResolve* realizing this strategy in Figure 6.17 (also found on the CD in the /linefollower/steering2/linefollower directory).

Figure 6.17 Strategy for Finding the Way Back to the Line (TurnResolve.java)

```
package linefollower;

import josx.robotics.Behavior;
import josx.platform.rcx.Motor;
import josx.platform.rcx.Sensor;
import josx.platform.rcx.Sound;
import josx.platform.rcx.Servo;

public class TurnResolve implements Behavior
{
  static final int HYSTERESIS = 4;
  Motor drive;
  Motor turn;
  Sensor rot;

  Servo servo;

  boolean running = false;
  boolean first = true;

  public TurnResolve(Motor drive, Motor turn, Sensor rot)
  {
    this.drive = drive;
    this.turn = turn;
    this.rot = rot;
    servo = new Servo (rot, turn);
  }

  public synchronized void action ()
  {
    if (running) {
        return;
    }
    Sound.beepSequence(); //indicates we have now started finding back
    running = true;
```

Continued

Figure 6.17 Continued

```
//curve back towards the line
try {
  turn.setPower (7);
  servo.rotateTo (1); //we will curve slightly towards the line
  synchronized (servo) {
    servo.wait();
  }
  drive.setPower(1);
  drive.forward();

  LineValueHolder lvh = LineValueHolder.getInstance();
  int light = -1;

  //assume we are spinning on white so go "straight" as
  //long as we are still on white
  do {
    light = lvh.getValue();
  } while (light > lvh.getWhite() - HYSTERESIS);

  //we should now be on black. Keep going until we reach white
  //again, and assume we crossed the line from the left side.
  do {
    light = lvh.getValue();
  } while (light < lvh.getBlack() + HYSTERESIS);

  //we should now be back on the left side of the line, so turn to
  //get onto the edge of the line again
  servo.rotateTo (2);
  drive.setPower(3);
  do {
    light = lvh.getValue();
  } while (light > lvh.getWhite() - HYSTERESIS);

  //we now hit the line from the left, so stop driving and
  //go straight
```

Continued

Figure 6.17 Continued

```
    } catch (InterruptedException ie) {
    } finally {
      servo.rotateTo (0);
      try {
        synchronized (servo) {
          servo.wait();
        }
      } catch (InterruptedException ie) {}
      drive.stop();
      running = false;
      notifyAll();
    }
  }

  /**
   * Control will be taken when the rotation sensor is showing 5.
   **/
  public boolean takeControl () {
     if (first) {
         first = false;
         return false;
     }
     if (running) {
       return true;
     } else {
       int current = rot.readValue();
       if (current >= 5) {
          return true;
       }
       return false;
     }
  }

  /**
```

Continued

Figure 6.17 Continued

```
   * Called if higher priority behavior wishes to take control.
   **/
  public synchronized void suppress () {
    while (running) {
      try {
        wait (200);
      } catch (InterruptedException ie) {
          //ignore
      }
    }
    Sound.systemSound(true, 3);
  }

}
```

As mentioned earlier, this is not particularly clever, and to be honest, it only works to some degree (the first solution, with limited turning, works much better). But maybe you can come up with something far superior. The main point of this example has been to introduce you to the subsumption architecture, which is a good way to separate your behavioral logic.

Debugging leJOS Programs

Debugging your RCX programs is not an easy task compared to debugging programs on a workstation. Basically, you have to rely on the visual feedback provided by the LCD, as well as audio feedback from the sound capabilities.

Using Sounds and the LCD

One of my favorite ways to debug leJOS programs is to use the RCX sound capabilities. You can play different sounds when turning right and left, or when some sensor reading is above a particular value. This is really useful, especially in combination with the LCD. By using different sounds before displaying a value on the LCD, you can easily know what the displayed value represents—a low tone might signify a light-sensor reading, or a high tone might alert you to a temperature reading. Just remember that you often need to halt the program

momentarily, using, for instance, *Button.VIEW.waitForPressAndRelease()*; otherwise, it will just be a mess of beeping and numbers flashing by in the display.

Exception Handling with leJOS

As you know from normal Java, exception handling is a great way to deal with errors in your programs. The throwing of exceptions in Java for signaling abnormal situations makes it possible to separate the functional logic of the code from the exception handling logic. This is the real benefit: separation of code into code for normal operation and code for error situations.

Normal Java exception handling applies to leJOS. For example, you can do constructs like the following:

```
try {
...
} catch (SomeException e) {
...
} finally {
}
```

And *throws* declarations can be put on method signatures, signaling to the method caller that it needs to handle the exceptions listed. Also, as in standard Java, only checked exceptions need to be handled with *try-catch* blocks, or declared in the *throws* clauses.

Of course, you can throw exceptions yourself, too, with constructs like:

```
throw new InterruptedException();
```

That allocated an object you cannot reclaim. If this code is executed often, you will eventually run out of memory. In these situations, you should allocate the *Exception* in a constructor and throw the same *Exception* instance when needed, like this:

```
public class SomeClass {
   InterruptedException ie = new InterruptedException();

   public void someMethod () throws InterruptedException
   {
      throw ie;
   }
}
```

On my wish-list is a construct like the one in JavaCard, where some exceptions have a static *throwIt(int code)* method declared. This method, when called, will throw a system-owned instance of that exception, which will eliminate pitfalls like the preceding one.

I must add though that in leJOS code, I usually do not throw many exceptions myself.

What about exceptions that are not caught inside your program, and travel all the way up to the leJOS runtime system? They will, of course, terminate your program—and while doing so, they will cause the buzzer to sound and show an exception number in the rightmost figure of the LCD, together with a method signature number in the main part of the display. See the "Using the leJOS Simulator" section for an explanation of how these numbers are to be interpreted.

Testing leJOS Programs

I find that testing leJOS programs can be quite entertaining. You think you have really come up with the perfect design and a really clever implementation and yet the outcome when run in the RCX is that your robot just circles in place, or a grabber opens the wrong way. The techniques using the RCX sound capabilities, mentioned previously are really helpful to determine what is going on. Remember, also, that turning a LEGO motor wire connection 90 degrees will reverse its operation. This often saves you from additional time-consuming downloads.

Before getting angry because the stupid thing misbehaves, remember that it only does as you instruct it! The RCX is a toy. It is supposed to be funny, creative, and, well, great to play with, and I really find that to be the case 100 percent of the time. So, relax and work those bugs out of your program.

Using the leJOS Simulator

Sometimes it is desirable to emulate the leJOS environment on your workstation. This often allows you to trace bugs more easily. The leJOS environment comes with two emulators, emu-lejos and emu-lejosrun, so do this:

```
emu-lejos -o <program-name>.bin <program-name>
emu-lejosrun -v <program-name>.bin
```

The *-v* option makes the emu-lejosrun output a bit more readable. For example, instead of outputting something like: *ROM call 1: 0x1946 (4096)*, it may instead resemble this: *set_sensor_active 0*. If exceptions are thrown, you will receive an output something akin to the following:

```
*** UNCAUGHT EXCEPTION/ERROR:
--  Exception class    : 14
--  Thread              : 1
--  Method signature   : 0
--  Root method sig.   : 0
--  Bytecode offset    : 8
```

Now, to really understand what is going on here you actually need to know which exception is represented as number 14, but this can easily be found if you put a *-verbose* option on the **emu-lejos** command (not the **emu-lejosrun** command); this will produce an output where (among other things) you can read:

```
...
Class 14: java/lang/InterruptedException
...
```

So, what was thrown upon you was an *InterruptedException* error. Now, if this was run on the RCX, the rightmost figure on the LCD would show 4, which is not the correct number (only one figure is used to display the number), illustrating the superior usability of the emulator.

More information is available, like which method threw the exception. Here the root method signature listing indicates the number 0, so you would again consult the verbose output, where you would see the signature numbers after the class numbers:

```
...
Signature 0: main([Ljava/lang/String;)V
...
```

So it was thrown from *main*, and the root method is also *main*. The root method refers to the method calling the guilty one. Finally, if you look at the bytecode, perhaps using the standard **javap** command, the bytecode offset will be 8, and you can thus exactly pinpoint the guilty statement.

If you use constructs like *Button.VIEW.waitForPressAndRelease()* in your program, you of course need to press the **View** button on your workstation. Unfortunately, such a button does not exist. Even worse, the present emulator does not define another button for the job. So, you need to eliminate such statements when running the emulator.

Note that the emulator uses a text-based interface, and it takes time to get the hang of it.

NOTE

On big endian machines like Sparc, you must use lejos instead of emu-lejos for creation of the binary package:

```
lejos -o <program-name>.bin <program-name>
emu-lejosrun -v <program-name>.bin
```

I actually prefer running my code in the RCX using the low-end debugging features described in Chapter 5, but that process might be bettered in the future. Andy Gombos' simulator, Simlink, for instance, contains (among other features) a graphical user interface (it's discussed in Chapter 7).

Summary

We covered a lot of ground in this chapter. The most important message to impose on you is to do good design first and then optimize as needed, although some design decisions have such a huge impact on the final implementation that you should keep in mind the constraints of the destination target while doing the design.

*String*s take up a lot of space, and manipulating them takes even more. This was explained in detail, and hopefully the benefits of creating your own classes in a mutable manner has since activated that little light bulb in your head, considering that most of the problems with *Strings* revolve about them being immutable.

We programmed a robot to follow the left edge of a black line, and tried different ways of making the robot either stay on that left side or find its way back to it should it happen to cross the line. This program was deliberately made in a way that could be optimized and the different optimizations explained.

You have also witnessed the useful robotic design technique named subsumption architecture. It was used to get the line-following robot back to the left side of the line in situations where the robot had crossed it. This architecture is not an alternative but rather a supplement to standard OO design. Its strength is that it allows you to separate your code into different robotic behaviors which can be individually programmed. The OO design on the other hand is more influenced with the internal design of those behaviors.

You have seen that debugging with leJOS on the RCX is tough work. The tools for debugging and testing are limited. Using sound and the LCD are probably the best ways at the moment, but better tools are on the horizon. How to use the leJOS emulator has also been explained, and even though its use is limited, it can sometimes save the day.

So, happy programming with leJOS, and play well.

Solutions Fast Track

Designing Java Programs to Run in leJOS

☑ When designing Java programs for leJOS, use the best OO design principles you know. It will make the final program much more maintainable.

- ☑ Pay attention to memory constraints even during the design phase. It is quite easy to do a beautiful OO design, which, when implemented, will use unnecessarily large amounts of memory.

An Advanced Programming Example Using leJOS

- ☑ The line-following type of robot can often become mired in a never-ending spin. Two techniques for avoiding that were presented.
- ☑ The subsumption architecture can be thought of as a design pattern for robotics programs.

Debugging leJOS Programs

- ☑ The best way to debug a leJOS program is to use the *Sound* and *LCD* classes in unison to provide you with feedback of the robot's state.
- ☑ Normal Java exception handling applies to leJOS, allowing you to separate code for normal operation and code for error situations.

Testing leJOS Programs

- ☑ When working out bugs, use emu-lejos and emu-lejosrun to emulate the leJOS environment on your PC. They use a text-based interface.
- ☑ When exceptions are output by the emulator, the output can be interpreted much more accurately than when displayed on the real RCX display.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: I get an *OutOfMemoryException* (indicated by a number 6 displayed on the RCX), but I do not believe I have allocated any memory. What might have caused this?

A: Remember, only the last exception class digit is displayed on the RCX, so it might be exception 16 or 26 that bit you. Another possibility is that even though you do not allocate memory explicitly, you might be doing it implicitly by using *String* arithmetic for instance.

Q: I keep getting *IllegalMonitorStateException*. What is the reason for this?

A: This is standard Java behavior. When you call *wait*, *notify*, or *notifyAll* on an object, you must own that object's monitor (for example, have synchronized access to it).

Q: The *Arbitrator* doesn't seem to function properly. Why is this happening?

A: This is probably a case of either your *suppress* or your *action* method not terminating. They must terminate. If you need continuous running behavior, create your own thread in the *action* method. Remember that you must be able to suspend that thread when the *suppress* method is called by the *Arbitrator*. See Figure 6.16 for an example of this.

Q: I have synchronized a method but it does not seem to work—I get some strange values for the static variables it updates. How can this be?

A: Your method is probably static. leJOS does not allow you to synchronize on static methods, as no instances of *java.lang.Class* are created.

Q: I have a thread instance which has terminated, and am trying to restart it. Why do I get an exception?

A: The exception you get is an *IllegalStateException*. It is defined in Java that threads cannot be restarted. It is thus not a leJOS-specific feature you are observing.