

Cole Sear in Shyamalan's *Sixth Sense* is not referring to dead bodies lying in front of him (for those who have not seen the movie). The five senses that most humans relate to are: touch, vision, balance, hearing, and taste or smell. In all cases our bodies have special sensory receptors that are placed on various parts of the body to enable sensing. For example the taste receptors are concentrated mostly on the tongue; the touch receptors are most sensitive on hands and the face and least on the back and on limbs although they are present all over the body, etc. Besides the difference in the physiology of each kind of receptors there are also different neuronal pathways and thereby sensing mechanisms built into our bodies. Functionally, we can say that each type of sensory system starts with the receptors which convert the thing they sense into electrochemical signals that are transmitted over neurons. Many of these pathways lead to the cerebral cortex in the brain where they are further *processed* (like, “Whoa, that jalapeño is hot!!”). The perceptual system of an organism refers to the set of sensory receptors, the neuronal pathways, and the processing of perceptual information in the brain. The brain is capable of combining sensory information from different receptors to create richer experiences than those facilitated by the individual receptors.

The perceptual system of any organism includes a set of *external* sensors (also called exteroceptors) and some *internal* sensing mechanisms (*interoceptors* or *proprioception*). Can you touch your belly button in the dark? This is because of proprioception. Your body's sensory

Proprioception

Sensing from within

Get something really delicious to eat, like a cookie, or a piece of chocolate, or candy (whatever you fancy!). Hold it in your right hand, and let your right arm hang naturally on your side. Now close your eyes, real tight, and try to eat the thing you are holding. Piece of cake! (well, whatever you picked to eat :-)

Without fail, you were able to pick it up and bring it to your mouth, right?

Give yourself a *Snickers moment* and enjoy the treat.

system also keeps track of the internal state of your body parts, how they are oriented, etc.

Sensing is an essential component of being a robot and every robot comes built with internal as well as external sensors. It is not uncommon, for example, to find sensors that are capable of sensing light, temperature, touch, distance to another object, etc. An example of internal sensing in robots is the measurement of movement relative to the robot's internal frame of reference. Sometimes also called *dead reckoning*, it can be a useful sensing mechanism that you can use to design robot behaviors.

Robots employ electromechanical sensors and there are different types of devices available for sensing the same physical quantity. For example, one common sensor found on many robots is a *proximity* sensor. It detects the distance to an object or an obstacle. Proximity sensors can be made using different technologies: infrared light, sonar, or even laser. Depending upon the type of technology used, their accuracy, performance, as well as cost vary: infrared (IR) is the cheapest, and laser is on the expensive side. Let's take a look at the perceptual system of your Scribbler robot starting with internal sensors.

Proprioception in the Scribbler

The Scribbler has three useful internal sensory mechanisms: *stall*, *time*, and *battery level*. When your program asks the robot to move it doesn't always imply that the robot is actually physically moving. It could be stuck against a wall, for example. The stall sensor in the Scribbler enables you to detect this. You have already seen how you can use time to control behaviors using the `MyroUtils.timeRemaining` and `MyroUtils.sleep` methods. Also, for most movement methods, you can specify how long you want that movement to take place (for example `forward(1, 2.5)` means full-speed forward for 2.5 seconds). Finally, it is also possible to detect battery power level so that you can detect when it is time to change batteries in the robot.

Time

All computers come built-in with an internal clock. In fact, clocks are so essential to the computers we use today that without them we would not have computers at all! Your Scribbler robot can use the computer's clock to sense time. It is with the help of this clock that we are able to use time in methods like `MyroUtils.timeRemaining`, `MyroUtils.sleep`, and other movement methods. Using only these facilities it is possible to define interesting automated behaviors.

Do This: Design a robot program for the Scribbler to draw a square (say with sides of 6 inches). To accomplish this, you will have to experiment with the movements of the robot and correlate them with time. The two movements you have to pay attention to are the rate at which the robot moves, when it moves in a straight line, and the degree of turn with respect to time. You can write a method for each of these:

```
private void travelStraight( double distance )
{
    // travel in a straight line for distance inches
    ...
}

private void degreeTurn( double angle )
{
    // Spin a total of angle degrees
    ...
}
```

Using the Sandbox project and your robot, experiment to figure out what the correlation is between the distance and the time for a given type of movement above and then use that to define the two methods described above. This correlation will be different for each robot, so it's important for you to use *your* robot for this experiment! For example, if a robot (hypothetical case) seems to travel at the rate of 25 inches/minute when you invoke the method `translate(1.0)`, then to travel 6 inches you will have to translate for a total of $6 * (60/25)$ seconds.

Conduct this experiment by moving your robot forward for varying amounts of time at the same fixed speed and recording the distance travelled. For example try moving the robot forward at speed 0.5 for 3, 4, 5, 6 seconds, and record the distance travelled by the robot for each of those times. You will notice a lot of variation in the distance even for the same set of commands. You may want to average those. Given this data, you can estimate the average distance your robot travels in one second. You can then define `travelStraight` as follows (assuming your calculation of inches per second was at half speed):

```
private void travelStraight( double distance )
{
    double inchesPerSecond = <your robot's value>;

    // travel in a straight line for distance inches
    robot.forward( 0.5, distance/inchesPerSecond );
}
```

Similarly you can also determine the time required for turning a given number of degrees. Try turning the robot at the same speed for varying amounts of time. Experiment how long it takes the robot to turn 360 degrees, 720 degrees, etc. Again, average the data you collect to get the number of degrees per second. Once you have figured out the details use them to write the `degreeTurn` function.

The following main method uses the above private methods to draw a square:

```
public void main()
{
    robot = new Scribbler("COM3");

    // transcribe a square of sides 6 inches
    for( int side = 0; side<4; side++ )
    {
        travelStraight( 6.0 );
        degreeTurn( 90.0 );
    }
}
```

```
        robot.close();  
    }
```

Do This: Create a BlueJ project named `Chapter_4_Sandbox`, then create program `drawFigure` containing the three methods described above (i.e., `travelStraight`, `degreeTurn`, and `main`). Run this program several times. It is unlikely that you will get a perfect square each time. This has to do with the calculations you performed as well with the variation in the robot's motors. They are not precise. Also, it generally takes more power to move from a still stop than to keep moving. Since you have no way of controlling this, at best you can only approximate this type of behavior. Over time, you will also notice that the error will aggregate. This will become evident in doing the next exercise.

Building on the ideas from the previous exercise, we could further abstract the robot's drawing behavior so that we can ask it to draw any regular polygon (given the number of sides and length of each side) with the method:

```
private drawPolygon( int numSides, double length )  
{  
    // draw a regular polygon with numSides sides, each side  
    // being length inches.  
    ...  
}
```

Then, we can write a regular polygon drawing robot program as follows:

Specifying the portName

Did you notice that this program does not invoke the `connect` method? Instead, it uses a feature of Myro/Java where you can specify the `portName` when the `Scribbler` object is created. This saves one line of code, which is not that big a deal.

The choice is yours – you can specify the `portName` when you create the `Scribbler` object, or create the object without the `portName` and invoke `connect` later.

```

public void main()
{
    // create the Scribbler object and connect
    robot = new Scribbler( "COM3" );

    // have the user enter the number of sides
    // and the length of each side
    int sides = MyroGUI.inputInteger("Enter # of sides");
    double len = MyroGUI.inputDouble("Enter side length");

    // draw the polygon
    drawPolygon( sides, len );

    // close the connection
    robot.close();
}

```

Do This: Modify program `drawFigure` from the previous exercise to include method `drawPolygon` and the modified `main`. (Hint: when drawing an n -sided regular polygon, turn $360/n$ degrees after drawing a side.) To test the program, first try drawing a square of sides 6 inches as in the previous exercise. Then try a triangle, a pentagon, hexagon, etc. Try a polygon with 30 sides of length 0.5 inches. What happens when you give 1 as the number of sides? What happens when you give zero (0) as the number of sides?

An Alternative Program Organization

Program `drawPicture` discussed in the previous section works as it should and is designed well. However, as is usually the case, there are other ways to design this

int vs. double, again

Why was the first parameter to method `drawPolygon` (i.e., `numSides`) an `int` and not a `double`? To help you think about this, consider the following questions: Does it make sense to draw a $3\frac{1}{2}$ sided polygon? No! On the other hand, does it make sense to draw a polygon with $8\frac{1}{2}$ inch sides? Yes!

When programming, values that can reasonably only be whole numbers (e.g. number of sides of a polygon) should be `ints`, and values that can be fractional (e.g., length of a side) should be `doubles`.

program that are equally good. Let's look at one of these alternate designs.

Before we begin it might be a good idea to review and highlight an important concept discussed in previous chapters. Have you noticed that there are two ways the classes we've written have used class `Scribbler`? Our programs (e.g., `drawPicture`) declare a `Scribbler` variable (called `robot`) that holds a `Scribbler` object created with a `new Scribbler()` statement. We called the relationship between class `drawPicture` and class `Scribbler` a "uses" relationship, because "class `drawPicture` *uses* a `Scribbler`".

In chapter 2, by contrast, we defined classes that extended (i.e., added methods to) class `Scribbler` (e.g., class `dancingScribbler` added some `yoyo` methods to class `Scribbler`). Even though we didn't use this term in that discussion, we call the relationship between class `dancingScribbler` and class `Scribbler` an "is-a" relationship, because "class `dancingScribbler` *is a* `Scribbler` with additional methods"¹.

Our alternate design for the `drawPicture` program will involve defining a new class called `calibratedScribbler` that extends class `Scribbler`, then using this new class in the main program class.

Class `calibratedScribbler` will extend `Scribbler` with public methods `travelStraight`, `degreeTurn`, and `drawPolygon`. Furthermore, we want to design `calibratedScribbler` so it will work with *any* `Scribbler` robot, not just yours.

Here is the class we have in mind:

```
public class calibratedScribbler extends Scribbler
{
    public void travelStraight( double distance )
    {
        double inchesPerSecond = <your robot's value>;
```

¹ In Java terminology, class `Scribbler` is the *base class*, and class `dancingScribbler` is a *derived class*.

```
        // travel in a straight line for distance inches
        forward( 0.5, distance/inchesPerSecond );
    }

    public void degreeTurn( double angle )
    {
        double degreesPerSecond = <your robot's value>;

        // spin for angle degrees
        turnLeft( 0.5, angle/degreesPerSecond );
    }

    public void drawPolygon( int numSides, double length )
    {
        // calculate number of degrees to turn for each side
        double degree = 360.0 / numSides;

        // draw the polygon
        for( int side=0; side<numSides; side++ )
        {
            travelStraight( length );
            degreeTurn( degree );
        }
    }
}
```

Do This: Create class `calibratedScribbler` in project `Chapter_4_Sandbox`. Test the class by creating an instance of `calibratedScribbler` and invoking each of the methods several times.

There is still a problem with this class – can you figure out what it is? Remember that we wanted our class to work with *any* robot. What is it about your implementation that is specific to your robot? Hopefully you realize that the values placed into variables `inchesPerSecond` and `degreesPerSecond` were selected based on *your* robot's behavior; each robot will most likely have different values for these. Thus, your class won't behave properly for all robots. Bummer ☹

Our approach to solving this problem will be to have each `calibratedScribbler` object have its own values for `inchesPerSecond` and `degreesPerSecond`; we'll also provide methods to change these values so a user can tailor a `calibratedScribbler` object to her or his specific robot.

Here's our revised class:

```
1   public class calibratedScribbler extends Scribbler
2   {
3       // declare instance fields
4       private double inchesPerSecond;
5       private double degreesPerSecond;
6
7       public void travelStraight( double distance )
8       {
9           // travel in a straight line for distance inches
10          forward( 0.5, distance/inchesPerSecond );
11      }
12
13     public void degreeTurn( double angle )
14     {
15         // spin for angle degrees
16         turnLeft( 0.5, angle/degreesPerSecond );
17     }
18
19     public void drawPolygon( int numSides, double length )
20     {
21         // calculate number of degrees to turn for each side
22         double degree = 360.0 / numSides;
23
24         // draw the polygon
25         for( int side=0; side<numSides; side++ )
26         {
27             travelStraight( length );
28             degreeTurn( degree );
29         }
30     }
31
32     public void setInchesPerSecond( double ips )
33     {
34         inchesPerSecond = ips;
35     }
```

```
36
37     public void setDegreesPerSecond( double dps )
38     {
39         degreesPerSecond = dps;
40     }
41 }
```

Let's take a look at the important changes we made to this class.

Lines 4 and 5: The declarations for variables `inchesPerSecond` and `degreesPerSecond` have been moved here, outside of all methods. Recall that variables declared within a method are called local variables and can only be used within the method in which they're defined. Variables declared at the class level (i.e., outside of all methods) are called *instance fields*. All methods within the class can directly use these variables.

Lines 10 and 16: Instance fields `inchesPerSecond` and `degreesPerSecond` are used in methods `travelStraight` and `degreeTurn`, just like they were in our earlier code.

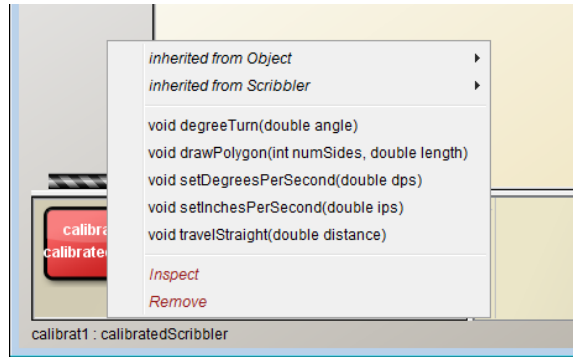
Lines 32-35, and 37-40: These lines define two new methods, named `setInchesPerSecond` and `setDegreesPerSecond`.

Lines 34 and 39: Values for instance fields `inchesPerSecond` and `degreesPerSecond` are set in methods `setInchesPerSecond` and `setDegreesPerSecond`, respectively.

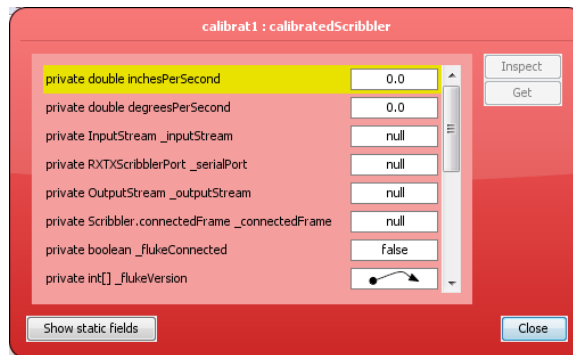
Do This: Modify class `calibratedScribbler` to include the above changes, compile it, and create an instance of the class. Invoke method `setInchesPerSecond`, passing it the correct value for your robot, then invoke method `setDegreesPerSecond`, passing it the correct value for your robot. Finally, invoke the other methods (e.g., `travelStraight`, `degreeTurn`, and `drawPolygon`) and verify that these methods behave properly for your robot.

To see what's happening here, let's use a feature of BlueJ that allows us to examine the values of instance fields.

Do This: Create a new instance of `calibratedScribbler`, then right-click on the object and select the `Inspect` menu item.

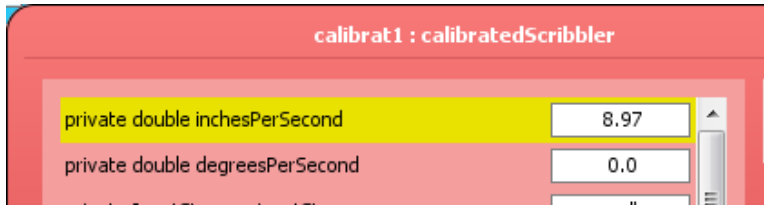


A window like the following will appear:

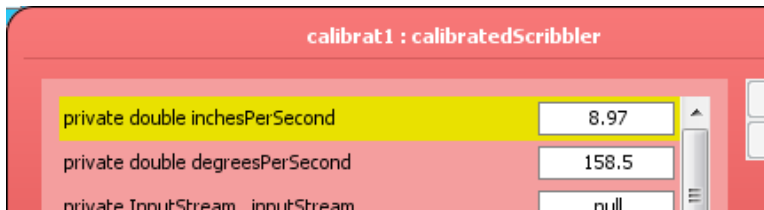


This window shows you the values in all instance fields of this object. The first two should be `inchesPerSecond` and `degreesPerSecond`, which should make sense because you defined these as instance fields in `calibratedScribbler`. But what the heck are the other instance fields? Remember that `calibratedScribbler` extends `Scribbler`, and all of these other instance fields are defined in class `Scribbler`. You will not need to ever work with (or even know about) any of these instance fields, so you should just ignore them (and accept our sincere apologies that you can even see them).

So, what are the values in `inchesPerSecond` and `degreesPerSecond` right now? They're both 0.0. Now invoke method `setInchesPerSecond` and note what happens to the value in field `inchesPerSecond` – it changes!



A similar change happens when `setDegreesPerSecond` is invoked:



Now think about what happens when method `travelStraight` is invoked. Recall that the following statement is executed within this method:

```
forward( 0.5, distance/inchesPerSecond );
```

What value will be used in the division? Whatever value is in instance field `inchesPerSecond`! And because the value in this field was set by `setInchesPerSecond` and contains the value specific to your robot, method `travelStraight` will correctly instruct your robot to move forward. How cool is that?!

Here is a program that instructs your robot to draw a polygon specified by the user, assuming the robot move forward at 8.97 inches per second and turns 158.5 degrees per second:

```
public class polygon
{
```

```
calibratedScribbler robot;

public void main()
{
    // create the Scribbler object and connect
    robot = new calibratedScribbler();
    robot.connect( "COM3" );

    // set the movement and turning rates
    robot.setInchesPerSecond( 8.97 );
    robot.setDegreesPerSecond( 158.5 );

    // have the user enter the number of sides
    // and the length of each side
    int sides = MyroGUI.inputInteger("Enter # of sides");
    double len = MyroGUI.inputDouble("Enter side length");

    // draw the polygon
    robot.drawPolygon( sides, len );

    // close the connection
    robot.close();
}
}
```

Before moving on there's one last issue we need to address. What happens if a user forgets to invoke `setInchesPerSecond` before invoking `travelStraight`? The answer is that `travelStraight` will use the initial value of field `inchesPerSecond`, which you recall is 0.0. What problems will this cause? Try it and see (i.e., create a new instance of `calibratedScribbler`, invoke `connect` to connect to your robot, then invoke `travelStraight`). You'll notice that your robot starts moving and will not stop! This is not a good behavior.

Why did this happen? When Java calculated how long to move forward it divided the distance by the value in `inchesPerSecond`, which was 0.0. In

Java, dividing a value by 0.0 produces the value infinity², so the robot is instructed to move forward for an infinite amount of time.

We should be able to define a better behavior for the robot in the event that a user forgets to invoke `setInchesPerSecond` before invoking `travelStraight`. We'll solve this problem by adding a special method called a *constructor* to the definition of `calibratedScribbler`:

```
public calibratedScribbler()
{
    inchesPerSecond = 8.97;
    degreesPerSecond = 158.5;
}
```

Constructor definitions are usually placed after the instance fields and before the first method, so in class `calibratedScribbler` you would place the above code between the declaration of instance field `degreesPerSecond` and the definition of method `travelStraight`.

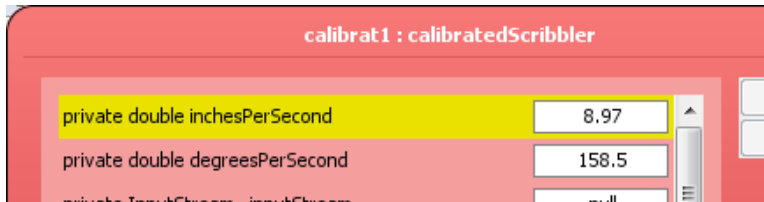
The constructor looks very similar to a method definition but there are some important differences. First, the constructor name *must* be the same as the class name; because this is a constructor for class `calibratedScribbler` the constructor must be named `calibratedScribbler`. Second, the constructor definition must not have the word `void` before the name.

It is not possible to directly invoke the constructor; instead, Java automatically invokes the constructor immediately after an instance of the class is created and before any other methods are invoked. Thus, the purpose of the constructor is to initialize the instance fields to whatever values are appropriate for the class. In this case we placed some non-zero values in fields `inchesPerSecond` and `degreesPerSecond`³.

² Oddly, when an `int` is divided 0 (i.e., integer 0) a runtime error occurs in Java.

³ The values we used in this example are for one of our robots. In this particular constructor you can use any “reasonable” values you’d like.

Do This: Add the constructor to class `calibratedScribbler` and create an instance of the class. Examine the instance fields of the object you just created (using the `Inspect` menu). The initial values of instance fields `inchesPerSecond` and `degreesPerSecond` should be the values the constructor placed in them (and not 0.0):



A Slight Detour: Random Walks

One way you can do interesting things with robot drawings is to inject some randomness in how long the robot does something. Java, and most programming languages, typically provide a library for generating random numbers. Generating random numbers is an interesting process in itself but we will save that discussion for a later time. Random numbers are very useful in all kinds of computer applications, especially games and in simulating real life phenomena (e.g., estimating how many cars might be entering an already crowded highway in the peak of rush hour).

Myro/Java provides two methods that generate random values:

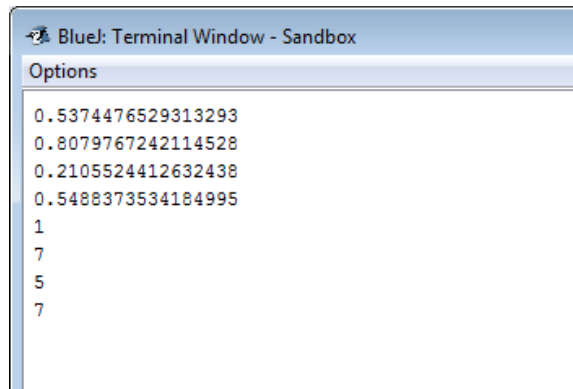
`MyroUtils.randomDouble()` Returns a random `double` between 0.0 (inclusive) and 1.0 (exclusive).

`MyroUtils.randomInt(A, B)` Returns a random `int` between A (inclusive) and B (inclusive).

Do This: Open the Sandbox project. In the codepad pane enter `System.out.println` statements that print the value returned by `MyroUtils.randomInt` and `MyroUtils.randomDouble`. For example, you might type the following in the codepad⁴:

```
System.out.println( MyroUtils.randomDouble() );
System.out.println( MyroUtils.randomDouble() );
System.out.println( MyroUtils.randomDouble() );
System.out.println( MyroUtils.randomDouble() );
System.out.println( MyroUtils.randomInt( 0, 10 ) );
System.out.println( MyroUtils.randomInt( 0, 10 ) );
System.out.println( MyroUtils.randomInt( 0, 10 ) );
System.out.println( MyroUtils.randomInt( 0, 10 ) );
```

The Terminal Window should contain something similar to:



```
Blue: Terminal Window - Sandbox
Options
0.5374476529313293
0.8079767242114528
0.2105524412632438
0.5488373534184995
1
7
5
7
```

As you can see, each time one of the random methods is invoked, a potentially different value is returned. How can this be used in a program? Consider the following program that has the Scribbler draw 10 random polygons:

```
public class randomPolygons
{
```

⁴When the cursor is in the codepad pane, pressing the “up arrow” key on the keyboard brings up the previous statement that was typed in the codepad. This feature is useful when you want to enter several statements that are similar.

```
// declare the Scribbler object
private calibratedScribbler robot;

public void main()
{
    // declare local variables
    int numSides, sideLength;

    // create the scribbler object and connect it
    robot = new calibratedScribbler();
    robot.connect( "COM3" );

    // set movement parameters for this robot
    robot.setInchesPerSecond( 9.87 );
    robot.setDegreesPerSecond( 158.5 );

    // draw 10 random polygons, allowing the user to
    // switch colors for each one
    for( int i=0; i<10; i++ )
    {
        // have user switch colors before continuing
        MyroGUI.tellUser
            ("Place a new pen in the Scribbler and hit OK");

        // generate the number of sides (between 3 and 8)
        // and length (between 2 and 6 inches) for a
        // polygon
        numSides = MyroUtils.randomInt( 3, 8 );
        sideLength = MyroUtils.randomInt( 2, 6 );

        // Print information so user knows what's
        // happening
        System.out.println("Polygon number " + i );
        System.out.println( numSides +
            "-sided polygon with length " + sideLength);

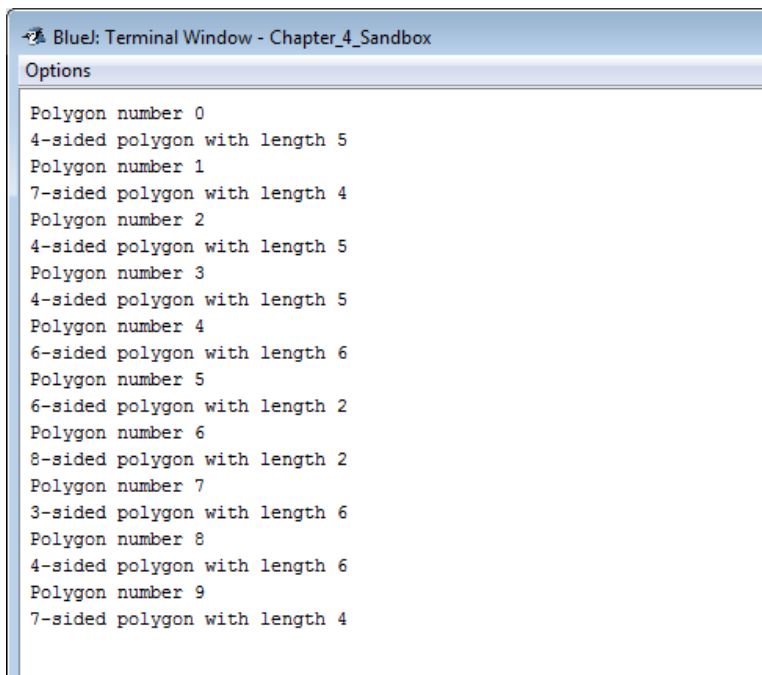
        // draw the polygon
        robot.drawPolygon( numSides, sideLength );
    }

    // finished, so close the connection
    robot.close();
}
```

```
}
```

Do This: Create program `randomPolygons`, shown above, in project `Chapter_4_Sandbox`, compile it, and execute it. Was it able to draw some random polygons? Hopefully it was.

When this program executes it prints some information about the polygons it's creating in the Terminal Window. For example, the Terminal Window should contain something like:



```
Blue: Terminal Window - Chapter_4_Sandbox
Options
Polygon number 0
4-sided polygon with length 5
Polygon number 1
7-sided polygon with length 4
Polygon number 2
4-sided polygon with length 5
Polygon number 3
4-sided polygon with length 5
Polygon number 4
6-sided polygon with length 6
Polygon number 5
6-sided polygon with length 2
Polygon number 6
8-sided polygon with length 2
Polygon number 7
3-sided polygon with length 6
Polygon number 8
4-sided polygon with length 6
Polygon number 9
7-sided polygon with length 4
```

Hopefully you notice that the number of sides of the polygons are indeed random numbers between 3 and 8, and the side lengths are random values between 2 and 6. Also note that not all random values in these ranges are selected; for example, in the execution shown above, no triangles were drawn, and no polygon had a length of 3. Does this suggest a problem or error? No – it simply demonstrates that the random values generated are really random. In

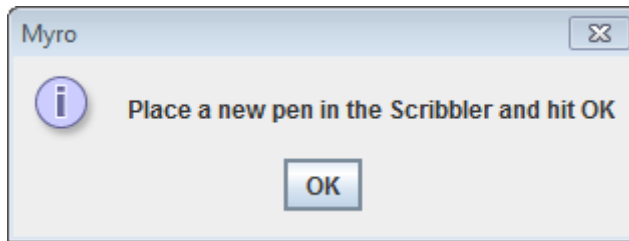
fact, if this program were executed again there is a good chance that one or more triangles would be drawn.

Also, note that during each execution of the for loop the value of `i` (the loop control variable) is printed by the first `System.out.println` statement. What value did variable `i` have when the loop was executed the first time? What value did it have during the second execution? What about the last execution? Do you see why for loops are sometimes called counting loops?

Finally, did you figure out what the statement:

```
MyroGUI.tellUser("Place a new pen in the Scribbler and hit OK");
```

does? If everything worked correctly the following dialog box should have appeared on the screen:



This Myro/Java method is useful to tell the user something and wait until she or he acknowledges the message by clicking the OK button.

Do This: Replace the statement mentioned above with the following:

```
MyroGUI.tellUser("New Pen Please", "Okey-Dokey");
```

What is different? What effect does the second parameter (e.g., “Okey-Dokey”) have?

As another example of the use of random values, consider the following program that instructs the robot to go on a random walk:

```
public class randomWalk
{
    // declare the Scribbler object
    private calibratedScribbler robot;

    public void main()
    {
        // create the scribbler object and connect it
        robot = new calibratedScribbler();
        robot.connect( "COM3" );

        // set the movement parameters for this robot
        robot.setInchesPerSecond( 9.87 );
        robot.setDegreesPerSecond( 158.5 );

        // perform a random walk for 20 steps
        for( int i=0; i<20; i++ )
        {
            robot.travelStraight(MyroUtils.randomInt( 0, 5 ));
            robot.degreeTurn( MyroUtils.randomInt( 0, 180 ));
        }

        // finished, so close connection
        robot.close();
    }
}
```

Do This: Create program `randomWalk` in project `Chapter_4_Sandbox` and execute it. Try playing with the parameters to the `MyroUtils.randomInt` methods to see how they affect the walk.

Do This: Write a Scribbler program of your own that exploits the Scribbler's movements to make random drawings. Make sure you generate drawings with at least three or more colors. Because of random movements, your robot is likely to run into things and get stuck. Help your robot out by picking it up and placing it elsewhere when this happens.

Back to time...

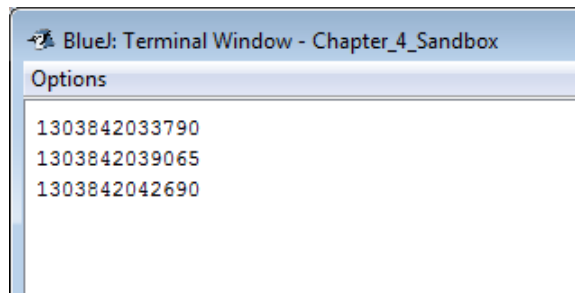
Most programming languages also allow you to access the computer's internal clock to keep track of time, or time elapsed (as in a stop watch), or in any other way you may want to make use of time (as in the case of the `MyroUtils.sleep` method). Java provides a simple method that can be used to retrieve the current time:

```
System.currentTimeMillis()
```

The value returned by `System.currentTimeMillis` is an integer that represents the number of milliseconds⁵ that have elapsed since some earlier time, whatever that is. Using the codepad, enter the following statement several times:

```
System.out.println(System.currentTimeMillis() );
```

You will notice that the difference in the values printed in the Terminal Window represents the real time in seconds. Here is a sample output, printing three times:



```
Blue: Terminal Window - Chapter_4_Sandbox
Options
1303842033790
1303842039065
1303842042690
```

Can you tell how much time passed between the first printing and the last? The time when the last `System.out.println` was executed was 1303842042690 msecs, and the time that the first `System.out.println`

⁵ A millisecond is one thousandth of a second. In other words, 1,000 milliseconds equals 1 second. Milliseconds is often abbreviated as msec or ms.

statement was executed was 1303842033790 msec. How much time elapsed? $1,303,842,042,690 - 1,303,842,033,790 = 8,900$, so 8,900 msec elapsed, which is 8.9 seconds.

Suppose the current time (in msec) is 1,303,842,033,790. What time will it be (in msec) 15.5 seconds from now? You can calculate this by adding 15,500 (which is the number of milliseconds in 15.5 seconds) to the current time. If you got 1,303,842,049,290 then you are correct!

One use of the current time is to determine how long some action takes. For example, here's a program that tests a user's reaction time to click the OK button on a dialog box:

```
public class reactionTime
{
    public void main()
    {
        // wait a random amount of time so the user
        // can't anticipate
        MyroUtils.sleep( MyroUtils.randomInt( 0, 10 ) );

        // remember the current time, then display the
        // dialog box
        long startTime = System.currentTimeMillis();
        MyroGUI.tellUser
            ("Press the OK button as fast as you can!");

        // look at the time now
        long endTime = System.currentTimeMillis();

        // calculate the user's reaction time, first
        // in milliseconds
        long timeMillis = endTime - startTime;

        // calculate the reaction time in seconds
        double reactionTime = timeMillis / 1000.0;

        // Show the user a dialog box with his/her
        // reaction time
        MyroGUI.tellUser("Your reaction time was " +
```

```
        reactionTime + " seconds");
    }
}
```

Do This: Create program `reactionTime` in `Chapter_4_Sandbox` and see how fast your reaction time is. You can also test your friends' reaction times to see who the fastest button pusher in your dorm is!

By the way, did you notice that variables `startTime`, `endTime`, and `timeMillis` were all type `long`? A `long` variable is an integer very similar to `int`, but whereas `ints` are integers between ± 2 billion, `longs` can be any integer between ± 9 quintillion! `System.currentTimeMillis()` returns a `long`, so we must place this value into a `long` variable.

Sensing Stall

We mentioned in the beginning of this chapter that the Scribbler has a way of sensing that it is stalled when trying to move. This is done by using the Myro/Java method `getStall`:

`getStall()` Returns `true` if the robot has stalled, `false` otherwise.

You can use this to detect that the robot has stalled and then control the behavior of your robot based on this. For example:

```
robot.forward( 0.75 );
while( MyroUtils.timeRemaining( 60 ) )
{
    if( robot.getStall() )
    {
        robot.stop();
        robot.beep( 440, 5 );
    }
}
robot.stop();
MyroGUI.tellUser( "That was fun!!" );
```

There is a new Java statement in this code – the `if`-statement. The action of the `if`-statement depends on whether the *condition*, which is enclosed in parentheses, is `true` or `false`. (Remember, method `getStall` returns either `true` or `false`, and in this program the value it returns then controls what the `if`-statement does next.) If the condition is `true` the `if`-statement executes the body of the `if`-statement, which are all the statements enclosed in curly braces; if the condition is `false` the `if`-statement skips over the body (i.e., the body is *not* executed).

Here's essentially what the above program does:

Start the robot going forward at $\frac{3}{4}$ speed.

If one minute isn't up yet, check to see if the robot has stalled; if it has stalled then stop the robot and beep. If one minute still isn't up, check to see if the robot has stalled; if it has stalled then stop the robot and beep. If one minute still isn't up, check to see if the robot has stalled; if it has stalled then stop the robot and beep. If one minute still isn't up, check to see if the robot has stalled; if it has stalled then stop the robot and beep. If one minute still isn't up, check to see if the robot has stalled; if it has stalled then stop the robot and beep. If one minute still isn't up, check to see if the robot has stalled; if it has stalled then stop the robot and beep. If one minute still isn't up, check to see if the robot has stalled; if it has stalled then stop the robot and beep. (And so on until one minute is up.)

When one minute is up, stop the robot and tell the user that this was fun.

Do This: Create a program called `stallDemo` in `Chapter_4_Sandbox` and enter the above code. (Note that you'll also have to include other statements in `main`, such as the statement to create the Scribbler object and the statement to close the connection at the end.) Execute the program several times,

placing your robot different distances from an obstacle. When does the “That was fun!!!” dialog box appear? What if you change the length of time from 60 seconds to 15 seconds? What happens if the robot doesn’t stall before the specified time? What happens if you remove the last “`robot.stop();`” statement *and* place the robot such that it won’t encounter an obstacle before the specified time?

Do This: Study the following program. How do you think your robot will behave when this program is executed? Now create a program called `headBanger` in `Chapter_4_Sandbox` that contains this code. Execute the program a few times, placing the robot different distances from a wall. Did the program behave like you thought it would?

```
robot.forward( 0.75 );
while( MyroUtils.timeRemaining( 60 ) )
{
    if( robot.getStall() )
    {
        robot.stop();
        robot.backward( 0.75, 1.0 );
        robot.forward( 0.75 );
    }
}
robot.stop();
MyroGUI.tellUser( "I have a headache!!" );
```

Do This: Write a Java program called `explorer` in `Chapter_4_Sandbox` that causes your robot to behave as follows: Have your robot start moving forward at $\frac{3}{4}$ speed. If the robot encounters an obstacle then have it stop, back up a bit, turn a bit, then start moving forward at $\frac{3}{4}$ speed again. Do this for 1 minute.

Demonstrate this program to some friends who are not in your course. Ask them for their reactions. You may notice that people tend to ascribe some form of *intelligence* to the robot. That is, your robot is sensing that it is stuck, and when it is, it stops moving forward and tries to find a direction it can

move that is unobstructed. We will return to this idea of *artificial intelligence* in a later chapter.

Sensing Battery Power Levels

Your Scribbler robot runs on 6 AA batteries. As with any electronic device, with use, the batteries will ultimately drain and you will need to replace them with fresh ones.

Myro/Java provides an internal battery-level sensing method, called `getBattery`, that returns the current voltage being supplied by the batteries. When the battery voltage level gets too low your robot will exhibit erratic behavior (e.g., it may not stop even though

your program has invoked the `stop` method). The battery voltage level of your Scribbler will vary between 0 and 9 volts (0 being totally drained). What “too low” means is something you will have to experiment and find out. The best way to do this is to record the battery level when you insert a fresh set of batteries. Then, over time, keep recording the battery levels as you go.

The Scribbler also has some built-in battery-level indicator lights. The red LED on the Scribbler robot remains lit when the power levels are high (or in the good range). It starts to flash when the battery level runs low. There is a similar LED on the Fluke dongle that flashes when the battery level is low. Can you find it? Just wait until the battery levels run low and you will see it flashing!

You can use battery-level sensing to define behaviors for robots so that they are carried out only when there is sufficient power available. For example, you could include the following `if`-statement in method `main` immediately after connecting to your robot:

```
if ( robot.getBattery() < 5.0 )
```

Disposing Batteries

Make sure that you dispose used batteries properly and responsibly. Batteries may contain hazardous materials like cadmium, mercury, lead, lithium, etc. which can be deadly pollutants if disposed in landfills. Find out your nearest battery recycling or disposal option to ensure proper disposal.

```
{
    MyroGUI.tellUser("Batteries are too low.");
}
```

This `if`-statement checks the voltage level of your batteries and if they're too low (i.e., below 5 volts) the user is notified about this by displaying a warning message.

Do This: Add the above code to one of your program, immediately after establishing the connection to your robot. Insert some old batteries into your robot and observe the behavior of your program.

Checking a Valid Connection

Another use of the `if`-statement is to check that the connection was successfully made to your robot. Perhaps you've executed a program where you accidentally used the wrong port name for your robot or you forgot to turn your robot on before running your program. (Or if this hasn't happened to you, perhaps you have a friend who has done this.) Right now the behavior of your program when this happens is very unfriendly. We can write a more user-friendly program by including the following `if`-statement immediately after connecting to your robot:

```
if( !robot.scribblerConnected() )
{
    MyroGUI.tellUser("Error connecting to robot");
    return;
}
```

Method `scribblerConnected` returns `true` if the connection was successfully made and `false` if it wasn't. The exclamation mark in front of the method invocation means "not" (as in "I like the smell of skunk. NOT!"). So this `if`-statement means "if the connection to your robot was not made then tell the user that there was a problem and terminate the program." The `return` statement returns from the method without executing any other statements, which effectively terminates the program.

In the next chapter we'll discuss the kinds of things that can be used in an `if`-statement condition, but from now on you should always include the above code to check that the connection was successfully made at the beginning of each program.

Summary

In this chapter you have learned about proprioception or internal sensory mechanisms. The Scribbler robot has three internal sensory mechanisms: time, stall, and battery-level. You have learned how to sense these quantities and also how to use them in defining automated robot behaviors. You also learned about random number generation and used it to define unpredictable robot behaviors. Later, we will also learn how to use random numbers to write games and to simulate natural phenomena. Sensing can also be used to define decision behaviors using conditional expressions in `if`-statements. You also learned the role of instance fields when defining a Java class, which is a very powerful and important concept in programming.

Myro/Java Review

```
MyroUtils.randomDouble()
```

Returns a random `double` value in the range 0.0 (inclusive) and 1.0 (exclusive).

```
MyroUtils.randomInt( int low, int high)
```

Returns a random `int` value in the range `low` (inclusive) and `high` (inclusive).

```
MyroGUI.tellUser( String message )
```

A dialog box with `message` is displayed along with an OK button. The user must click the OK button.

```
MyroGUI.tellUser( String message, String button-text )
```

A dialog box with `message` is displayed along with a button containing `button-text`. The user must click the button.

```
getStall()
```

Returns `true` if the robot is stalled when trying to move, `false` otherwise.

```
getBattery()
```

Returns the current battery power level (in volts). It can be a number between 0 and 9 with 0 indicating no power and 9 being the highest. There are also LED power indicators present on the robot. The robot behavior becomes erratic when batteries run low. It is then time to replace all batteries.

Java Review

```
System.currentTimeMillis()
```

This returns the number of milliseconds since some point in time in the past. The returned value is a `long` integer.

Exercises

1. Write a robot program to make your Scribbler draw a five point star. [Hint: Each vertex in the star has an interior angle of 36 degrees.]
2. Experiment with Scribbler movement commands and learn how to make it transcribe a path of any given radius. Write a program to draw a circle of any input diameter.
3. Write a program to draw other shapes: the outline of a house, a stadium, or create art by inserting pens of different colors. Write the program so that the robot stops and asks you for a new color.
4. If you had an open rectangular lawn (with no trees or obstructions in it) you could use a Zamboni-like strategy to mow the lawn. Start at one end of the lawn, mow the entire length of it along the longest side, turn around and mow the entire length again, next to the previously mowed area, etc. until you are done. Write a program for your Scribbler to implement this strategy (make the Scribbler draw its path as it goes).

5. Enhance the random drawing program from this chapter to make use of speech. Make the robot, as it is carrying out random movements, to speak out what it is doing. As a result you will have a robot artist that you have created!
6. Rewrite your program from the previous exercise so that the random behavior using each different pen is carried out for 30 seconds.
7. The Myro library also provides a function called, `randomNumber()` that returns a random number in the range 0.0 and 1.0. This is similar to the function `random()` from the Python library `random` that was introduced in this chapter. You can use either based on your own preference. You will have to import the appropriate library depending on the function you choose to use. Experiment with both to convince yourself that these two are equivalent.
8. In reality, you only need the function `random()` to generate random numbers in any range. For example, you can get a random number between 1 and 6 with `randRange(1, 6)` or as shown below:

```
randomValue = 1 + int(random()*6)
```

The function `int()` takes any number as its parameter, truncates it to a whole number and returns an *integer*. Given that `random()` returns values between 0.0 (inclusive) and 1.0 (exclusive), the above expression will assign a random value between 1..5 (inclusive) to `randomValue`. Given this example, write a new function called `myRandRange()` that works just like `randrange()`:

```
def myRandRange(A, B):
```

```
    # generate a random number between A..B
    # (just like as defined for randrange)
```

9. What kinds of things can your robot talk about? You have already seen how to make the robot/computer speak a given sentence or phrase. But the robot can also "talk" about other things, like the time or the weather.

One way to get the current time and date is to import another Python library called `time`:

```
>>> from time import *
```

The time module provides a function called `localtime` that works as follows:

```
>>> localtime()  
(2007, 5, 29, 12, 15, 49, 1, 149, 1)
```

`localtime` returns all of the following in order:

1. year
2. month
3. day
4. hour
5. minute
6. seconds
7. weekday
8. day of the year
9. whether it is using daylight savings time, or not

In the example above, it is May 29, 2007 at 12:15pm and 49 seconds. It is also the 1st day of the week, 149 day of the year, and we are using daylight savings time. You can assign each of the values to named variables as shown below:

```
year, month, day, ..., dayOfWeek = localtime()
```

Then, for the example above, the variable `year` will have the value 2007; `month` will have the value 5, etc. Write a Python program that speaks out the current date and time.

