Most people associate the personal computer (aka the PC) revolution with the 1980's but the idea of a personal computer has been around almost as long as computers themselves. Today, on most college campuses, there are more personal computers than people. The goal of One Laptop Per Child (OLPC) Project is to "provide children around the world with new opportunities to explore, experiment, and express themselves" (see www.laptop.org). Personal robots, similarly, were conceived several decades ago. However, the personal robot 'revolution' is still in its infancy. The picture on the previous page shows the Pleo robots that are designed to emulate behaviors of an infant Camarasaurus. The Pleos are marketed mainly as toys or as mechatronic "pets". Robots these days are being used in a variety of situations to perform a diverse range of tasks: like mowing a lawn; vacuuming or scrubbing a floor; entertainment; as companions for elders; etc. The range of applications for robots today is limited only by our imagination! As an example, scientists in Japan have developed a baby seal robot (shown on the opposite page) that is being used for therapeutic purposes for nursing home patients.

Your Scribbler robot is your personal robot. In this case it is being used as an educational robot to learn about robots and computing. As you have already seen, your Scribbler is a rover, a robot that moves around. Such robots have become more prevalent in the last few years and represent a new dimension of robot applications. Roaming robots have been used for mail delivery in large offices and as vacuum cleaners in homes. Robots vary in the ways in which they move about: they can roll about like small vehicles (like the lawn mower, Roomba, Scribbler, etc.), or even ambulate on two, three, or more legs (e.g. Pleo). The Scribbler robot moves on three wheels, two of which are powered. In this chapter, we will get to know the Scribbler in some more detail and also learn about how to use Myro/Java methods to control its behavior.

The Scribbler Robot: Movements

In the last chapter you were able to use the Scribbler robot through Myro/Java to carry out simple movements. You were able to start the BlueJ environment, create a Scribbler object and connect it to your robot, and then were able to make it beep, give it a name, and move it around using a joystick. By inserting

a pen in the pen port, the Scribbler robot is able to trace its path of movements on a piece of paper placed on the ground. It would be a good idea to review all of these tasks to refresh your memory before proceeding to look at some more details about controlling the Scribbler.

The Paro Baby Seal Robot. Photo courtesy of National Institute of Advanced Industrial Science and Technology, Japan (paro.jp).

If you hold the Scribbler in your hand and take a look at it, you will notice that it has three wheels. Two of its wheels (the big ones on either side) are powered by motors. Go ahead turn the wheels and you will feel the resistance of the motors. The third wheel (in the front) is a free wheel that is there for support only. All the movements the Scribbler performs are controlled through the two motor-driven wheels. In Myro/Java there are several methods to control the movements of the robot. The method that directly controls the two motors is the `motors` method:

```
motors( left, right )
```

`left` and `right` can be any value in the range `[-1.0 to +1.0]` and these values control the left and right motors, respectively. Specifying a negative value moves the motors/wheels backwards and positive values move it forward. Thus, the method invocation:

```
motors( 1.0, 1.0 )
```

will cause the robot to move forward at full speed, and the invocation:

```
motors( 0.0, 1.0 )
```

will cause the left motor to stop and the right motor to move forward at full speed resulting in the robot turning left. Thus by giving a combination of left and right motor values, you can control the robot's movements. Myro/Java has

also provided a set of methods for commonly used movements that are easier to remember and use. Some of them are listed below:

```
forward( speed )
backward( speed )
turnLeft( speed )
turnRight( speed )
stop()
```

Another version of these methods takes a second argument, an amount of time in seconds:

```
forward( speed, seconds )
backward( speed, seconds )
turnLeft( speed, seconds )
turnRight( speed, seconds )
```

Providing a number for `seconds` in the above methods specifies how long that movement will be carried out. For example, if you wanted to make your robot traverse a square path, you could issue the following sequence of method invocations:

```
forward( 1, 1 )
turnLeft( 1, .3 )
forward( 1, 1 )
turnLeft( 1, .3 )
forward( 1, 1 )
turnLeft( 1, .3 )
forward( 1, 1 )
turnLeft( 1, .3 )
```

Of course, whether you get a square or not will depend on how much the robot turns in 0.3 seconds. There is no direct way to ask your robot to turn exactly 90 degrees, or to move a certain specified distance (say, 2 ½ feet). We will return to this later.

You can also use the following movement commands to translate (i.e. move forward or backward), or rotate (turn right or left):

```
translate( speed )
rotate( speed )
```

Additionally, you can specify, in a single method invocation, the amount of translation and rotation you wish:

```
move( translate_speed, rotate_speed )
```

In all of these methods, `speed` can be a value between `[-1.0 to +1.0]`.

You can probably tell from the above list that there are a number of redundant methods (i.e. several methods can be invoked to make the same movement). This is by design. You can pick and choose the set of movement methods that appear most convenient to you. It would be a good idea at this point to try out these methods on your robot.

Do This: Start BlueJ, open the Sandbox project, create a Scribbler object and connect it to your robot. Place your robot on the floor, making sure there are a few feet of open space in front of it. Then try out the following movement methods on your Scribbler[1]:

```
motors( 1, 1 )
motors( 0, 0 )
```

Observe the behavior of your robot. Specifically, notice if it does (or doesn't) move in a straight line after invoking the first `motors` method. You can make the robot carry out the same behavior by invoking the following methods:

```
move( 1.0, 0.0 )
stop()
```

Go ahead and try these. The behavior should be exactly the same. Next, try making the robot go backwards using any of the following methods:

---

[1] The method invocation motors(1,1) is done in the Sandbox by right-clicking on the Scribbler object and selecting the `motors` method from the menu. In the dialog box that appears, enter 1 in both of the boxes and click OK.

```
motors( -1, -1 )
move( -1, 0 )
backward( 1 )
```

Again, notice the behavior closely. In rovers, precise movement, like moving in a straight line, is difficult to achieve. This is because two independent motors control the robot's movements. In order to move the robot forward or backward in a straight line, the two motors would have to issue the exact same amount of power to both wheels. While this technically feasible, there are several other factors that can contribute to a mismatch of wheel rotation. For example, slight differences in the mounting of the wheels, different resistance from the floor on either side, etc. This is not necessarily a bad or undesirable thing in these kinds of robots. Under similar circumstances even people are unable to move in a precise straight line. To illustrate this point, you can try the experiment shown at right.

Do humans walk straight?

Find a long empty hallway and make sure you have a friend with you to help with this. Stand in the center of the hallway and mark your spot. Looking straight ahead, walk about 10-15 paces without looking at the floor. Stop, mark your spot and see if you walked in a straight line.

Next, go back to the original starting spot and do the same exercise with your eyes closed. Make sure your friend is there to warn you in case you are about to run into an object or a wall. Again, note your spot and see if you walked in a straight line.

For most people, this experiment will result in a variable movement. Unless you really concentrate hard on walking in a straight line, you are most likely to display similar variability as your Scribbler. Walking in a straight line requires constant feedback and adjustment, something humans are quite adept at doing. This is hard for robots to do. Luckily, roving does not require such precise movements.

Do This: Review all of the other movement methods listed above and try them out on your Scribbler. Again, note the behavior of the robot from each of these methods.

classes and objects, again

The difference between a class and an object is a difficult concept to grasp. An analogy that might be helpful is the difference between a blueprint and a house.

A blueprint is the design of a house, but it is not a house. Architects and engineers develop the blueprint based on the needs of the person or business. They specify room dimensions, where doors are to be located, where sinks are placed, etc. You can look at a blueprint and imagine opening a door, for example, but you can't actually open a door on a blueprint.

A house is constructed from the blueprint. You can open a door in a house, use the sink, etc.

A Java class is the "blueprint" of an object. The class is designed to do something, and it is used to construct an object. Actions are actually done on the object.

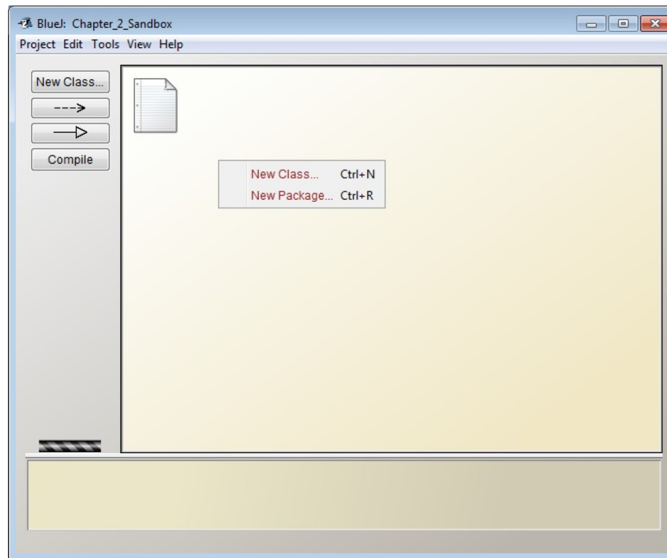
Defining New Methods

Trying out individual Scribbler methods using the Sandbox project is a nice way to get to know your robot's basic features and we'll continue to use this each time we want to try something new. However, making your robot carry out more complex behaviors requires a series of several method invocations. Having to invoke these methods over and over while the robot is operating can get tedious.

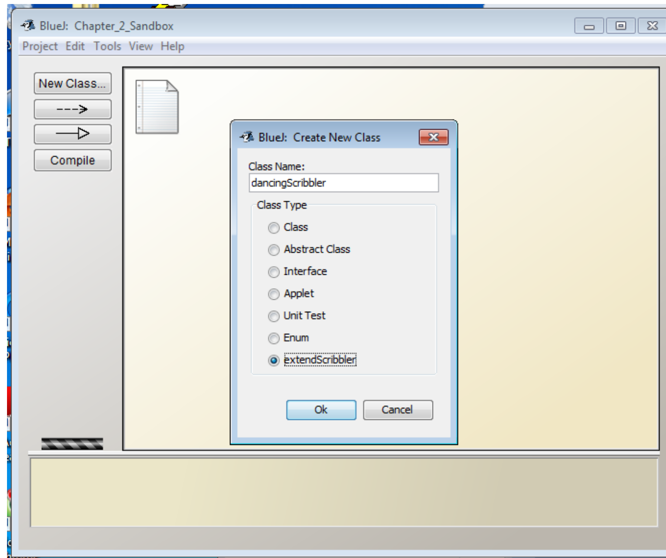
Luckily Java provides a way for us to define new methods. In Java, methods can only be defined in a class. In this chapter we'll go over how to define a new class that adds methods to an existing class; specifically we'll define some new classes that have all Scribbler methods, plus some new ones that we'll define.

Suppose we want to add a method called `yoyo` that moves the robot forward and then backward (like a yoyo). We'll define a new class (let's call it dancingScribbler) that is a Scribbler but with one more method: yoyo. Because this is your first venture into defining a new class we'll go through the process step by step.

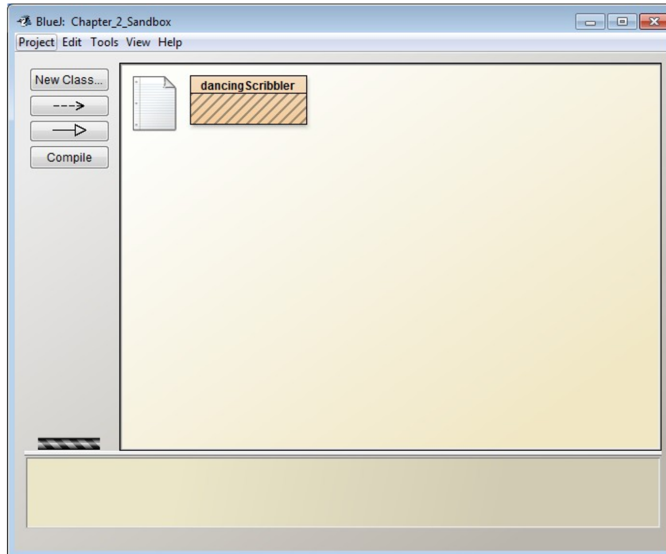
Start BlueJ and open the Chapter_2_Sandbox project using the `Project->Open Project…` menu. (If there are other projects open you might want to close them to keep your screen from getting too cluttered.) Right-click anywhere in the main pane in the project and select "New Class…" from the menu.



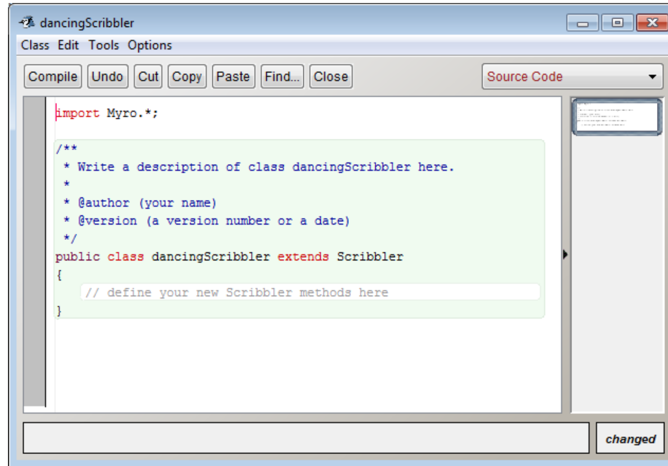
A dialog box will appear asking you for the name and type of class you want to create. Let's call our new class dancingScribbler. Enter dancingScribbler in the text field, then select the "extendScribbler" radio button, and click OK.



A class named dancingScribbler will appear in the project pane.



Double-click the dancingScribbler icon (or right-click it and select OpenEditor from the menu). A window will appear that contains the Java code for this new class.



We'll look at what all of this means shortly, but for now edit the class so it contains the following text (putting in your name and today's date where indicated):

```
import Myro.*;

/**
 * Adds some dancing methods to a Scribbler.
 *
 * @author Douglas Harms
 * @version April 10, 2011
 */
public class dancingScribbler extends Scribbler
{
    // make the robot go back and forth like a yoyo
    public void yoyo()
    {
        forward( 1 );
        backward( 1 );
        stop();
    }
```

```
}
```

When you're finished, click the Compile button, located toward the top left of the editor window. When you do this the computer checks to make sure you entered valid Java (we'll talk much more about what valid Java looks like soon enough). If everything is fine, the status area at the bottom of the window will display "Class compiled-no syntax errors", otherwise the status area will contain an indication about what the problem was and the line with the error will be highlighted. If this happens to you and you can't figure out what's wrong, ask your instructor or one of the lab assistants for help.

Scribbler Tip:

Remember that your Scribbler runs on batteries and with time they will get drained. When the batteries start to run low, the Scribbler may exhibit erratic movements. Eventually it stops responding. When the batteries start to run low, the Fluke board's red LED light starts to blink. This is your signal to replace the batteries.

Once your class compiles without errors you can instantiate it (i.e., create an object of your new class) by right-clicking on the dancingScribbler icon and selecting `new dancingScribbler()`.

Right-clicking on the object just created shows you all of the methods available on that object. `yoyo` is apparently the only available method. However, if you select the "inherited from Scribbler" menu item you'll see that all of the Scribbler methods are also available! So an instance of `dancingScribbler` can do everything a Scribbler can do, plus it can yoyo!

Connect the `dancingScribbler` object to your robot and spend some time experimenting with the `yoyo` method as well as some of the Scribbler methods. When you're finished be sure to invoke the `close` method.

Did the yoyo method work? In other words, did your robot move forward and then backward? In fact, it did, but the backward movement was very brief and may have been difficult for you to observe.

Can you see the problem? Before reading the next paragraph take a minute or two to consider the instructions we gave the robot in the `yoyo` method.

The problem is that right after telling the robot to move backward we told it to stop! The amount of time it moved backward was very short, perhaps not even noticeable. One way to solve this problem is to tell the computer to wait before continuing. Here's how we can do this:

```
public void yoyo()
{
    forward( 1 );
    MyroUtils.sleep( 0.5 );
    backward( 1 );
    MyroUtils.sleep( 0.5 );
    stop();
}
```

`MyroUtils.sleep( seconds )` is a Myro/Java method that causes the computer to pause for the specified number of seconds before continuing.

Do This: Edit the definition of the yoyo method to include the above changes, then recompile it and test it. How did yoyo behave? You should have seen your robot move forward for ½ second, then backward for ½ second, then stop.

Can you think of other ways to solve the problem we had with our first definition of yoyo? (Think about other forward and backward methods.) Spend some time trying some of your ideas.

Congratulations!! You have now written your first Java class! Go ahead and pat yourself on the back and have a soda (or pop, depending on where you live) to celebrate.

What's in a Java Class

Let's briefly look at what's in the `dancingScribbler` class. As a reminder, here's the definition of the class (with line numbers for easy reference in the discussion):

```
1   import Myro.*;
2
3   /**
4    * Adds some dancing methods to a Scribbler.
5    *
6    * @author Douglas Harms
7    * @version April 10, 2011
8    */
9   public class dancingScribbler extends Scribbler
10  {
11      // make the robot go back and forth like a yoyo
12      public void yoyo()
13      {
14          forward( 1 );
15          MyroUtils.sleep( 0.5 );
16          backward( 1 );
17          MyroUtils.sleep( 0.5 );
18          stop();
19      }
20  }
```

Line 1: This statement indicates that we're going to use Myro/Java methods and other definitions in this class. Without this statement Java would not know about class Scribbler, Scribbler methods (e.g., forward, backward), etc. All of our classes will start with this statement.

Line 9: This statement indicates that we're defining a new class called `dancingScribbler` and that it extends (i.e., adds methods to those defined in) class Scribbler. The word `public` is required; we'll discuss reasons for this later in the course.

Line 12: This statement indicates that the class will have a method named yoyo and that it doesn't have any parameters. The words `public` and `void` are required; we'll look at what they mean soon.

Lines 10 and 20, 13 and 19: All definitions within a class, and all statements within a method definition, must be enclosed in curly braces (i.e., `{ }`). We'll also see that curly braces are used in a number of other ways in Java programs.

Lines 14-18: These are the statements that will be executed when the yoyo method is invoked. This is often called the *body* of the method.

Lines 3-8, and 11: All of these lines contain comments. Comments are completely ignored by the Java compiler, so it doesn't matter what's inside them. So why include comments? So other programmers (and you!) can know what's going on in the class. We'll see different styles of comments during the course.

There are two ways to specify comments in a Java program. In the first comment specification, the comment starts with the characters `/*` and ends with the characters `*/` . In this program the comment starts at the beginning of line3 and ends on line 8. Note that this kind of comment can span multiple lines. (The asterisks at the beginning of lines 4-7 are not required.)

In the other specification, the comment begins with two slash characters and ends at the end of the line. Line 11 contains this kind of comment.

What kind should you use? It's really up to you!

Adding Parameters to Methods

Take a look at the definition of the `yoyo` method above and you will notice the use of parentheses, **()**. You have also used other methods with parentheses in them and probably can guess their purpose. Methods can specify certain *parameters* (or values) by placing them within parentheses. For example, all

of the movement methods, with the exception of `stop`, have one or more parameters that you specify to indicate the speed of the movement or the amount of time to move.

We could have specified the speed of forward and backward movement as a parameter when we defined method yoyo. Or we could have specified the amount of time to yoyo as a parameter.

Here are definitions for three additional yoyo methods, each specifying one or more parameters:

```
public void yoyo1( double speed )
{
    forward( speed, 1.0 );
    backward( speed, 1.0 );
}

public void yoyo2( double waitTime )
{
    forward( 1.0, waitTime );
    backward( 1.0, waitTime );
}

public void yoyo3( double speed, double waitTime )
{
    forward( speed, waitTime );
    backward( speed, waitTime );
}
```

In yoyo1 we specify the speed of the forward and backward movement as a parameter. When yoyo1 is invoked, the speed to use is passed to the method. For example, if you wanted to move at half speed you could invoke yoyo1 as:

```
yoyo1( 0.5 );
```

Similarly, method yoyo2 is specified with the wait time as a parameter, and yoyo3 is specified with both speed and wait time as parameters.

Incidentally, the word `double` in the parameter list specifies that the parameter is a number that can have a fractional component, such as 0.86. In Java terminology, `double` is a *type*. Why did the designers of Java choose the confusing word double to mean this? It's not really important to know why – just accept it as part of Java. In the next chapter we'll look at other types of numbers.

Do This: Add these yoyo methods to class `dancingScribbler`, recompile the class, and test all of the yoyo methods.

By the way, did you notice that none of the new yoyo methods invoked `stop` at the end? Why is `stop` not necessary in these methods?

Have you also noticed how similar all four yoyo methods are? Each one consists of an invocation of `forward` followed by an invocation of `backward`. The only difference between the yoyos is the parameters passed to these method invocations that determine the speed and duration of the movements.

`yoyo3` is the most general of the yoyos because both speed and duration are parameters. Can you think of a way that the other yoyos could be implemented by simply invoking `yoyo3` once?

In fact, this is not only possible but a good programming technique. Consider the following definition of `yoyo` which, as you recall, instructs your robot to yoyo at a speed of 1 for 1 second:

```
public void yoyo()
{
    yoyo3( 1.0, 1.0 );
}
```

Do This: Implement `yoyo1` and `yoyo2` so they each invoke `yoyo3`. Edit `dancingScribbler` with the new definitions of `yoyo`, `yoyo1`, and `yoyo2`, compile it, and test. Do all four yoyos behave correctly? They should.

Another Example – Dance Competition!

Suppose you want to enter your robot in the upcoming robot dance competition. The competition rules specify that you must be able to have your robot perform three dance routines:

1. Shimmy once then shuffle once
2. Shuffle twice then shimmy once
3. Shimmy once, then shuffle twice, then shimmy three more times

Shimmying consists of the following moves:

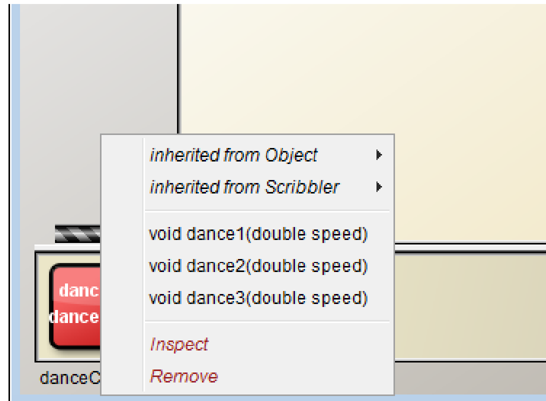
1. Rotate left at a specified speed for 1 second
2. Rotate right at the same speed for 2 seconds
3. Rotate left at the same speed for 1 second

Shuffling consists of the following moves:

1. Move forward at a specified speed for 1 second
2. Move backward at the same speed for 2 seconds
3. Move forward at the same speed for 1 second

Let's define a Java class named `danceCompetition` that extends the Scribbler class with three new methods named `dance1`, `dance2`, and `dance3`. Each of these methods will have one parameter that determines the speed of movement.

According to the rules of the dance competition, when the judges test your robot they should see only three methods defined on the `danceCompetition` object (`dance1`, `dance2`, and `dance3`), as shown here:



Let’s get started. As with most problems, there are multiple ways we could approach this. For example, we could define method `dance2` like this:

```
public void dance2( double speed )
{
    // shuffle once
    forward( speed, 1.0 );
    backward( speed, 2.0 );
    forward( speed, 1.0 );

    // shuffle again
    forward( speed, 1.0 );
    backward( speed, 2.0 );
    forward( speed, 1.0 );

    // shimmy once
    turnLeft( speed, 1.0 );
    turnRight( speed, 2.0 );
    turnLeft( speed, 1.0 );    }
```

This approach would work, but do you see any issues with it? You’ve hopefully noticed that the three statements to have the robot shuffle are repeated twice. Imagine the implementation of method `dance3` that has the robot shimmy a total of four times. Wouldn’t it be great if there were a way to implement the dance methods that didn’t involve writing the same code over and over? In fact there is a way to do this; can you think what it is?

Here's an alternate approach: let's define two additional methods in `danceCompetition` – one to shuffle and one to shimmy. Once this is done then the dance methods can be implemented by simply invoking `shuffle` and `shimmy` appropriately.

Here's a definition of methods `shimmy` and `shuffle`:

```
public void shimmy( double speed )
{
    turnLeft( speed, 1.0 );
    turnRight( speed, 2.0 );
    turnLeft( speed, 1.0 );    }

public void shuffle( double speed )
{
    forward( speed, 1.0 );
    backward( speed, 2.0 );
    forward( speed, 1.0 );
}
```

Method `dance2` can now be now implemented as:

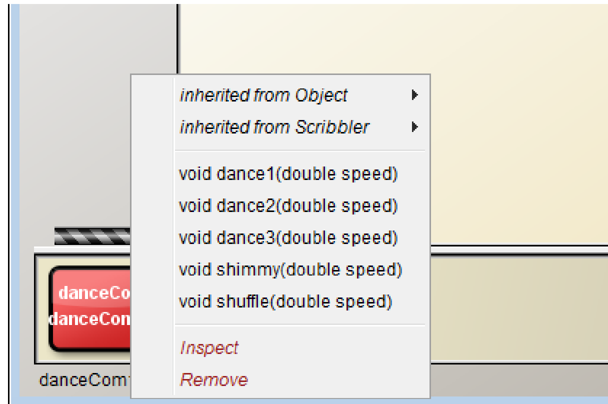
```
public void dance2( double speed )
{
    // shuffle twice
    shuffle( speed );
    shuffle( speed );

    // shimmy once
    shimmy( speed );
}
```

Do This: Create the `danceCompetition` class in the Chapter_2_Sandbox project, defining methods `shimmy`, `shuffle`, `dance1`, `dance2`, and `dance3`. Compile and test your class. Does your robot dance as it should?

Unfortunately there's still one problem with the `danceCompetition` class you've implemented: the judges will disqualify you. Why? Remember the

rule mentioned earlier about only having the three dance methods displayed. When invoking methods of your `danceCompetition` object, the following is shown:



The judges will never invoke methods `shimmy` or `shuffle`. In fact, the only reason we defined these methods was to make our lives easier when we implemented the three dance methods. Said another way, methods `shimmy` and `shuffle` are not meant to be seen by anyone using class `danceCompetition`, instead they were defined only to be used by us when we implemented the dance methods.

We can solve this problem by making methods shimmy and shuffle private methods instead of public methods:

```
private void shimmy( double speed )
{
    turnLeft( speed, 1.0 );
    turnRight( speed, 2.0 );
    turnLeft( speed, 1.0 );    }

private void shuffle( double speed )
{
    forward( speed, 1.0 );
    backward( speed, 2.0 );
    forward( speed, 1.0 );
}
```

Note that all we have done here is change the word `public` to `private` in the definitions of these two methods. Once we make this change, methods `shimmy` and `shuffle` cannot be invoked by a user of a `danceCompetition` object (they are not even in the object's menu!), but other methods of class `danceCompetition` can still invoke them.

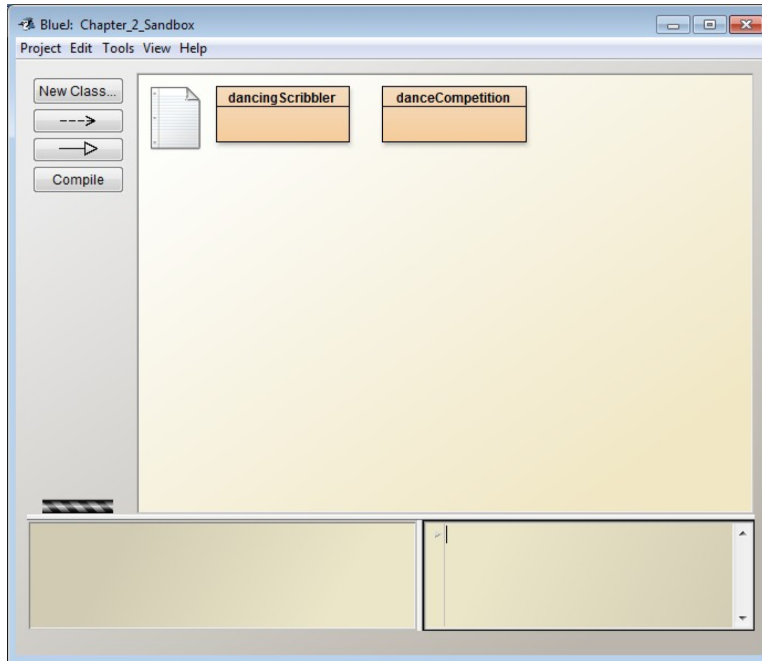
Do This: Make methods `shimmy` and `shuffle` private methods in your `danceCompetition` class, recompile the class and test it. When you invoke methods of a `danceCompetition` object the only available methods should be `dance1`, `dance2`, and `dance3`, which will make the judges happy. Will your robot get first place in the dance competition without being disqualified? It should!

BlueJ Codepad

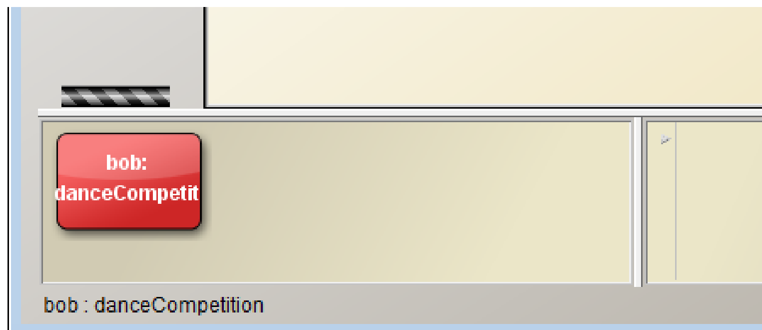
Almost all of your interactions with Java classes and objects so far have been using BlueJ's GUI (Graphical User Interface); you've instantiated a class (i.e., created an object) by right-clicking on the class icon and selecting a "new" menu item; you've invoked an object's method by right-clicking on the object and selecting the method you want to invoke. The only time you've needed the keyboard was to enter a parameter value for a method invocation. Using the BlueJ GUI is a wonderful and easy way to get to know a Java class by experimenting with its methods.

In addition to the GUI, BlueJ has something called the codepad that is a text-based way to experiment with and test methods. The codepad can be shown in the BlueJ window by selecting the `View->Show Code Pad` menu.

Do This: Open the Chapter_2_Sandbox project where you created your `dancingScribbler` and `danceCompetition` classes, then show the codepad (`View->Show Code Pad` menu). You should see the following:



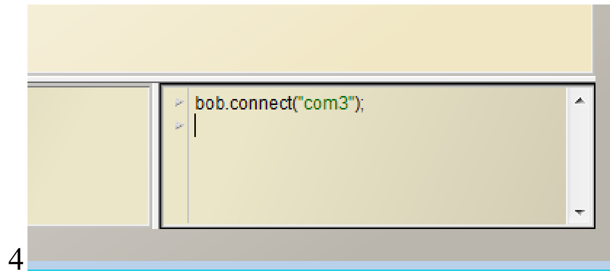
The codepad is the new window pane located at the lower righthand side of the window. Create a `danceCompetition` object and name this object `bob` (i.e., when you create the object, enter the word "bob" (without quotes) in the dialog box that appears). You should see a `danceCompetition` object at the bottom of the window named `bob`:



In the codepad, type the following:

```
bob.connect("com3");
```

Be sure to include the semicolon at the end, and put the portname of your robot in quotes.



4

You have now connected the `bob` object to your robot. Now tell your robot to do the first dance at half speed by entering the following in the codepad:

```
bob.dance1( 0.5 );
```

Your robot should perform the first dance routine. Now invoke other methods on object `bob`, such as `dance2`, `forward`, `joyStick`, `camera`, etc. When you're finished experimenting be sure to invoke the `close` method.

So, what exactly does the statement "`bob.dance1(0.5);`" do? Well, it tells Java to invoke the `dance1` method on the object named `bob`, passing the value 0.5 as the parameter. This has the exact behavior as using the GUI (i.e., right-clicking on the `bob` object, selecting method `dance1` from the menu, and entering 0.5 in the dialog box that appears).

Think About It: What happens if you enter "dance1(0.5);" in the codepad? You'll see that Java can't find method `dance1`. The problem is that you didn't tell Java which object to invoke the `dance1` method on. For example, imagine creating two `danceCompetition` objects (let's name then `bob` and `carol`):

Connecting `bob` to the robot on port com3 is done with:

```
bob.connect("com3");
```

and connecting `carol` to the robot on port com12 is done with:

```
carol.connect("com12");
```

How do we tell `carol` to perform `dance2` at half speed and `bob` to perform `dance3` and ¾ speed? The following should do the trick:

```
carol.dance2( 0.5 );
bob.dance3( 0.75 );
```

Why don't you need to specify the object name when using the GUI? Because it is clear which object you want to use by which object you right-click on. If you want to invoke a method on object `carol` you right-click on `carol` and select a method; if you want to invoke a method on object `bob` you right-click on `bob` and select a method.

As you should have noticed when you defined the various methods in this chapter, Java programs are defined using textual statements. The statements you enter in the codepad are real Java statements, and we'll be seeing a lot more of these starting in the next chapter.

Guided by Automated Controls

Earlier we agreed that a robot is a "mechanism guided by automated controls". You can see that by defining methods that carry out more complex movements, you can create methods for many different kinds of behaviors. The methods make up the programs you write, and when they are invoked on the robot, the robot carries out the specified behavior. This is the beginning of being able to define automated controls for a robot. As you learn more about the robot's capabilities and how to access them via Myro/Java methods, you can design and define many kinds of automated behaviors.

Summary

In this chapter, you have learned several Myro/Java methods that make a robot move in different ways. You also learned how to define methods for new robot behaviors by extending the Scribbler class. You learned that public methods in a class can be invoked from both inside and outside the class, but private methods can only be invoked from within the class.

While you have learned some very simple methods to control the robot, you have also learned some important concepts in computing that enable the building of more complex behaviors. While the concepts themselves are simple enough, they represent a very powerful and fundamental mechanism employed in almost all software development. In later chapters, we will provide more details about writing methods and also how to structure parameters that customize individual method invocations. Make sure you do some or all of the exercises in this chapter to review these concepts.

Myro/Java Review

`backward( speed )`
Move backwards at `speed` (value in the range -1.0…1.0).

`backward( speed, seconds )`
Move backwards at `speed` (value in the range -1.0…1.0) for a time given in `SECONDS`, then stop.

`forward( speed )`
Move forward at `speed` (value in the range -1.0..1.0).

`forward( speed, time )`
Move forward at `speed` (value in the range -1.0…1.0) for a time given in seconds, then stop.

`motors( left, right )`
Turn the left motor at `left` speed and right motor at `right` speed (value in the range -1.0…1.0).

`move( translate, rotate )`
Move at the `translate` and `rotate` speeds (value in the range -1.0…1.0).

`rotate( speed )`
Rotates at `speed` (value in the range -1.0…1.0). Negative values rotate right (clockwise) and positive values rotate left (counter-clockwise).

`stop()`
Stops the robot.

`translate( speed )`
Move in a straight line at `speed` (value in the range -1.0…1.0). Negative values specify backward movement and positive values specify forward movement.

`turnLeft( speed )`
Turn left at `speed` (value in the range -1.0…1.0)

`turnLeft( speed, seconds )`
Turn left at `speed` (value in the range -1.0..1.0) for a time given in seconds, then stops.

`turnRight( speed )`
Turn right at `speed` (value in the range -1.0..1.0)

`turnRight( speed, seconds )`
Turn right at `speed` (value in the range -1.0..1.0) for a time given in seconds, then stops.

`MyroUtils.sleep( time )`
Pause for the given amount of `time` seconds. `time` can be a decimal number.

Java Review

Exercises

1. Compare the robot's movements in the method invocations `turnLeft(1)`, `turnRight(1)` and `rotate(1)` and `rotate(-1)`. Closely observe the robot's behavior and then also try these methods:

```
motors( -0.5, 0.5 )
motors( 0.5, -0.5 )
motors( 0, 0.5 )
motors( 0.5, 0 )
```

Do you notice any difference in the turning behaviors? The `rotate` commands make the robot turn with a radius equivalent to the width of the robot (distance between the two left and right wheels). The `turn` command causes the robot to spin in the same place.

2. Insert a pen in the scribbler's pen port and then issue it command to go forward for 1 or more seconds and then backwards for the same amount. Does

the robot travel the same distance? Does it traverse the same trajectory? Record your observations.

3. Measure the length of the line drawn by the robot in Exercise 2. Write a function `travel( distance )` to make the robot travel the given distance. You may use inches or centimeters as your units. Test the function on the robot a few times to see how accurate the line is.

4. Suppose you wanted to turn/spin your robot a given amount, say 90 degrees. Before you try this on your robot, do it yourself. That is, stand in one spot, draw a line dividing your two feet, and then turn 90 degrees. If you have no way of measuring, your turns will only be approximate. You can study the behavior of your robot similarly by invoking turn methods and making them wait a certain amount of time. Try and estimate the wait time required to turn 90 degrees (you will have to fix the speed) and write a method to turn that amount. Using this method, write a behavior for your robot to transcribe a square on the floor (you can insert a pen to see how the square turns out).

5. Generalize the wait time obtained in Exercise 3 and write a method called `degreeTurn( degrees )`. Each time it is invoked, it will make the robot turn the specified degrees. Use this method to write a set of instructions to draw a square.

6. Using the methods `travel` and `degreeTurn`, write a method to draw the Bluetooth logo (See Chapter 1, Exercise 9).

7. Choreograph a simple dance routine for your robot and define methods to carry it out. Make sure you divide the tasks into re-usable moves and as much as possible parameterize the moves so they can be used in customized ways in different steps. Use the building block idea to build more and more complex series of dance moves. Make sure the routine lasts for at least several seconds and it includes at least two repetitions of the entire sequence. You may also make use of the `beep` method you learned from the last chapter to incorporate some sounds in your choreography.

8. Record a video of your robot dance and then dub it with a soundtrack of your choosing. Use whatever video editing software accessible to you. Post the video online on sites like YouTube to share with friends.

9. Lawn mower robots and even vacuuming robots can use specific *choreographed* movements to ensure that they provide full coverage of the area to be serviced. Assuming that the area to be mowed or cleaned is rectangular and without any obstructions, can you design a behavior for your Scribbler to provide full coverage of the area? Describe it in writing. [Hint: Think about how you would mow/vacuum yourself.]

Potential sidebars:

Inheritance? Object-oriented terminology?

Pre-post conditions?

Documentation

case sensitivity? Syntactic pickiness (e.g., semicolons)

BNF-style definitions?