

If you think of your robot as a creature that acts in the world, then by programming it you are essentially building the creature's brain. The power of computers lies in the fact that the same computer (or the robot) can be supplied a different program (or brain) to make it behave like a different creature. For example, a program like Firefox or Explorer makes your computer behave like a web browser. But switching to your Media Player, the computer behaves as a DVD or a CD player. Similarly, your robot will behave differently depending upon the instructions in the program that you have requested to run on it. In this chapter we will learn about the structure of Java programs and how you can organize different robot behaviors as programs.

The world of robots and computers, as you have seen so far, is intricately connected. You have been using a computer to connect to your robot and then controlling it by invoking various methods. Most of the methods you have used so far come from the Myro/Java library which is specially written for easily controlling robots. The programming language we are using to do the robot programming is Java. Java is a general purpose programming language. By that we mean that one can use Java to write software to control the computer or another device like a robot through that computer. Thus, by learning to write robot programs you are also learning how to program computers. Our journey into the world of robots is therefore intricately tied up with the world of computers and computing. We will continue to interweave concepts related to robots and computers throughout this journey. In this chapter, we will learn more about robot and computer programs and their structure.

Basic Structure of a Robot Brain (aka Java Program)

If you're like most college students today you probably don't remember a time without computers; you've grown up knowing about a variety of programs from word processors to web browsers to interactive games. Each program you've used is designed to do exactly one thing, and when you want your computer to do a particular thing you simply execute the appropriate program. For example, if you want to type a term paper you execute a word processor; the word processor program is designed to do exactly one thing: be

a word processor, and there is only one thing you ask it to do when you execute (i.e., invoke) it: be a word processor.

This is quite different from what we've done with Java so far. When you wanted to do something with a Scribbler object, was there only one method available? No! There were several dozen methods you could invoke. You didn't have the option of simply telling the Scribbler to "be a scribbler," but rather you had to choose from many options (e.g., move forward, backward, etc.). Unlike the word processor that only does one thing, the Java classes we've discussed so far do many things.

Is it possible to write a Java program (i.e., a Java class that does only one thing)? Yes!! *A Java program is a class with exactly one method available for invocation*; when that method is invoked on an object, the program will "do its thing." In the coming weeks we will talk about and write many programs, and each one will have a single thing that it does.

So, what does a Java program look like? Let's start by looking at a simple program that instructs your robot to move forward at half speed for 1 second, then turn counterclockwise at $\frac{3}{4}$ speed for 2 seconds, then play a 440 Hz tone for 1 second. It's not a very complicated behavior, but it's a start.

```
1  import Myro.*;
2  /**
3   * This is our first program.  It doesn't do much - just
4   * moves the robot around a bit.  But it's a start.
5   *
6   * @author Douglas Harms
7   * @version 14 April 2011
8   */
9  public class firstProgram
10 {
11     private Scribbler robot;
12
13     public void main()
14     {
```

```
15 // create a Scribbler object and connect it to my
16 // robot
17 robot = new Scribbler();
18 robot.connect( "COM3" ); // Use your robot's portname
19
20 // now have the robot do the moves and make the sound
21 robot.forward( 0.5, 1.0 );
22 robot.turnLeft( 0.75, 2.0 );
23 robot.beep( 1.0, 440 );
24
25 // We're finished, so close the connection
26 robot.close();
27 }
28 }
```

Do This: Create a class named `firstProgram` in BlueJ project `Chapter_3_Sandbox`; select `scribblerMain` as the class type¹. Edit the class so it contains the above program, being sure to enter your name and today's date in the appropriate places, and to use your robot's portname in the connect method invocation. Compile class `firstProgram`, instantiate it, and invoke the `main` method. Your robot should perform a simple song and dance.

Now that you've executed this program and observed its behavior,

Java Programs

During this course when you are asked to “write a Java program to do such-and-such” what we mean is “write a Java class that has exactly one `public` method, named `main`, and when `main` is invoked it will do such-and-such.”

There is nothing particularly special about the name `main`; as far as Java is concerned the method could be given any name, such as `makeItSoNumberOne` or `justDoIt`. However, the tradition in Java is to name this method `main`, and we'll respect that tradition.

¹ When creating a new class in BlueJ, the dialog box presents several templates for your new class. Selecting the appropriate one makes your life easier since it already contains much of the code for your new class. If you select the wrong class type that's OK – you'll just have to do a bit more editing.

let's take a closer look at what's in it.

Line 1: This statement allows the program to reference all definitions from Myro/Java. As mentioned in Chapter 2, all of the classes we create in this course will start with this statement.

Line 9: This statement starts the definition of our class, named `firstProgram`. Note that, unlike the classes discussed in Chapter 2, the `firstProgram` class *does not* extend `Scribbler`.

Line 11: This statement declares a variable named `robot` that can contain a `Scribbler` object. `private` indicates that this variable can only be used within the class being defined (i.e., class `firstProgram`).

Lines 13-27: These statements define a method named `main`. Because `main` is a `public` method it can be invoked from outside the class.

Line 17: This statement creates a new instance of class `Scribbler` and places this newly created `Scribbler` object in the variable named `robot`. The action of this statement is equivalent to instantiating a class in BlueJ by right-clicking on the class icon and selecting one of the `new` items from the menu.

Lines 18-26: These statements invoke various methods on the `Scribbler` object in variable `robot`. Note that the syntax of these statements is identical to what you entered in the BlueJ codepad in the previous chapter.

Dance Competition – Revisited

Let's write a Java program to do the third dance routine at half speed from the dance competition discussed in the previous chapter. Recall that the third dance routine consists of one shimmy, then two shuffles, then three more shimmies. Before reading ahead spend a few minutes thinking about how you would design this program.

We'll discuss two programs that exhibit the specified behavior. Here's the first one:

```
import Myro.*;

public class competition_one
{
    // declare the Scribbler object
    private Scribbler robot;

    public void main( )
    {
        // create a Scribbler object and connect to my robot
        robot = new Scribbler();
        robot.connect( "COM3" );

        // have the robot dance
        System.out.println( "About to dance..." );
        shimmy( 0.5 );
        shuffle( 0.5 );
        shuffle( 0.5 );
        shimmy( 0.5 );
        shimmy( 0.5 );
        shimmy( 0.5 );
        System.out.println( "...Done dancing!" );

        // disconnect from the robot
        robot.close();
    }

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

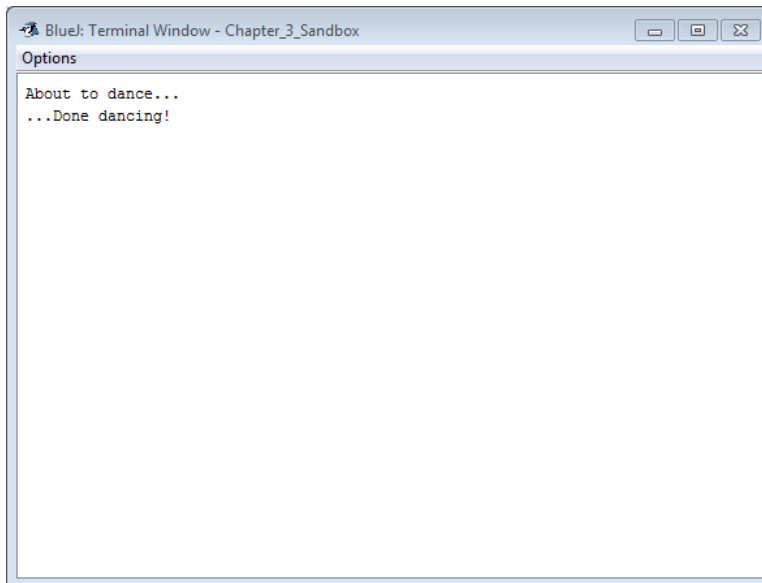
    private void shuffle( double speed )
    {
        robot.forward( speed, 1.0 );
        robot.backward( speed, 2.0 );
        robot.forward( speed, 1.0 );
    }
}
```

```
}
```

Do This: Create a class named `competition_one` in project `Chapter_3_Sandbox`, enter the above program in it, compile it and execute it. Did your robot do the third dance routine? Did anything else happen?

Did you notice that methods `shuffle` and `shimmy` were `private` methods? Our reasons for doing this are the same as what was discussed in Chapter 2. Remember too that a Java program must have only one `public` method (named `main`), so any other methods you define should be `private`.

You probably also noticed the `System.out.println` statements in this program². Did you notice what they did when you executed your program? If everything worked correctly a window should have been displayed when you executed your program that looks similar to:



² Actually, `System.out.println` is a method invocation, not a Java statement.

`System.out.println` statements are a good way to print information about your program as it's executing; we'll use them throughout the course.

You may be wondering why we couldn't use the `dance3` method here that we defined in the `danceCompetition` class in Chapter 2. If you look closely, the above program uses class `Scribbler`, and not class `danceCompetition`. We couldn't invoke method `dance3` because this method isn't defined for class `Scribbler`.

Let's rewrite this program to use a `danceCompetition` instead of a `Scribbler`:

```
import Myro.*;

/**
 * This program has the robot perform the third dance routine
 * at half speed for the dance competition.
 *
 * @author Douglas Harms
 * @version 16 April 2011
 */
public class competition_two
{
    // declare the danceCompetition object
    private danceCompetition robot;

    public void main()
    {
        // create a danceCompetition object and connect it
        robot = new danceCompetition();
        robot.connect( "COM3" );

        // have the robot dance
        System.out.println( "About to dance..." );
        robot.dance3( 0.5 );
        System.out.println( "...Done dancing!" );

        // disconnect from the robot
        robot.close();
    }
}
```

```
}
```

Notice that method `main` is relatively simple – all it does is connect to the robot (which is now a `danceCompetition` object instead of a `Scribbler` object), then invoke the `dance3` method, then `close` the connection. Most of the work is done in method `dance3` that we wrote in Chapter 2! Reusing code is a very important concept in software development.

There is one slight problem that needs to be addressed before you can enter and execute this program. The problem is that Java doesn't know about class `danceCompetition`. Why not? Because Java only knows about classes defined in your current project. You defined class `danceCompetition` in BlueJ project `Chapter_2_Sandbox`, but `competition_two` is defined in project `Chapter_3_Sandbox` so it can't use `danceCompetition`.

This problem is most easily solved by creating a `danceCompetition` class in `Chapter_3_Sandbox`. You could enter the code all over again from scratch, but here's a slightly easier way:

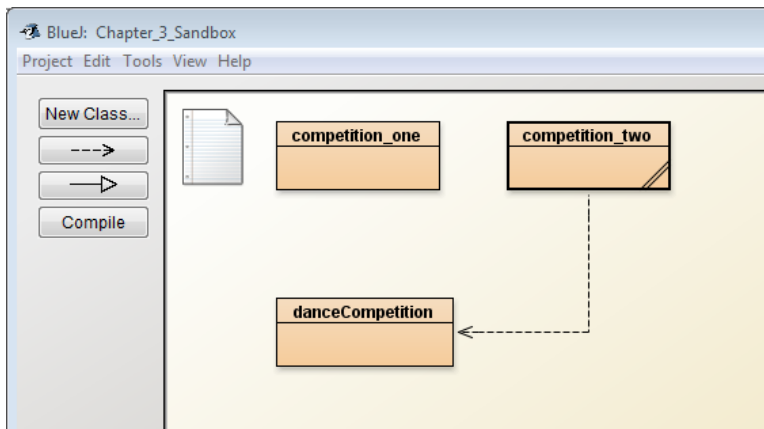
1. Open the `Chapter_2_Sandbox` project
2. Open the `danceCompetition` class by double-clicking on the class icon
3. Select all the code in `danceCompetition` and copy it to the clipboard
4. Create a new class called `danceCompetition` in `Chapter_3_Sandbox`
5. Open the new `danceCompetition` class, delete all of the text in this class, and paste the clipboard. The class should now be an identical copy of the `danceCompetition` class from `Chapter_2_Sandbox`.
6. Compile `danceCompetition`

Once class `danceCompetition` has been defined in project `Chapter_3_Sandbox`, program `competition_two` can be successfully compiled and executed.

Do This: Create class `danceCompetition` in project `Chapter_3_Sandbox`, then create program `competition_two`. Compile both classes and execute

`competition_two`. Your robot should successfully perform the third dance routine.

You might notice that BlueJ draws an arrow from class `competition_two` to class `danceCompetition` in the `Chapter_3_Sandbox` window:



This is a “uses arrow” and is a visual indication that class `competition_two` “uses” class `danceCompetition`. In complex projects that involve many classes, these arrows are good ways to see the relationships between the various classes.

Speaking Java

We have launched you into the world of computers and robots without really giving you a formal introduction to the Java language. In this section we provide more details about the language. What you know about Java so far is that it is needed to control the robot. The robot methods you invoke are integrated into Java by way of classes and methods defined in the `Myro/Java` package. Java comes with several other useful packages that we will learn throughout this course. If you need to access the methods and classes provided by a package, all you have to do is import the package (e.g., `import Myro.*;`).

The packages themselves consist of classes and methods that provide the basic building blocks for any Java program. Typically, a programming language (and Java is no exception) includes a set of pre-defined classes and methods, and a mechanism for defining additional classes and methods. You have already seen several examples of class and method definitions and indeed have written some of your own.

When defining a new class or method, you have to give the new class or method a *name*. Names are a critical component of programming and Java has rules about what forms a name.

What's in a name?

A name in Java must begin with either an alphabetic letter (a-z or A-Z) or the underscore (i.e. `_`) and can be followed by any sequence of zero or more letters, digits, or underscores. For example,

```
iRobot  
myRobot  
jitterBug  
jitterBug2  
my2cents  
my_2_cents
```

are all examples of valid Java names. Additionally, another important part of the syntax of names is that Java is *case sensitive*. That is the names `myRobot` and `MyRobot` and `myrobot` are distinct names as far as Java is concerned. Once you name something a particular way, you have to consistently use that exact case and spelling from then on.

Well, so much about the syntax of names, the bigger question you may be asking is *what kinds of things can (or should) be named?*

So far you have seen that names are used to represent classes (e.g., `danceCompetition`) and methods (e.g., `dance3`, `shimmy`). By giving methods a name you have a way of defining new methods. Names can also be used to

represent other things in a program. For instance, you may want to represent a quantity, like speed or time, by a name. In fact, you did so in Chapter 2 when defining the `yoyo` method in class `dancingScribbler`, which is also shown below:

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

Methods can take parameters that help customize what they do. In the above example, you can invoke the `yoyo` method in the following ways:

```
yoyo( 0.8, 2.5 );
yoyo( 0.3, 1.5 );
```

The first invocation performs the `yoyo` behavior at speed 0.8 for 2.5 seconds whereas the second one specifies 0.3 and 1.5 for speed and time, respectively. Thus, by parameterizing the method with those two values you are able to produce similar but varying outcomes. This idea is similar to the idea of mathematical functions: $\text{sine}(x)$ for example, computes the sine of whatever value you supply for x .

There has to be a way of defining the method in the first place that makes it independent of specific parameter values. That is where names come in. In the definition of the method `yoyo` you have named two parameters (the order you list them is important): `speed` and `waitTime`. Then you have used those names to specify the behavior that makes up that method. That is the invocations `forward` and `backward` use the names `speed` and `waitTime` to specify whatever the speed and wait times are included in the method invocation. Thus, the names `speed` and `waitTime` represent or designate specific values in this Java method.

Names in Java can represent methods as well as values. What names you use is entirely up to you. It is a good idea to pick names that are easy to read, type,

and also appropriately designate the entity they represent. What name you pick to designate a method or value in your program is very important, for you. For example, it would make sense if you named a method `turnRight` so that when invoked, the robot turned right. It would not make any sense if the robot actually turned left instead, or worse yet, did the equivalent of the yoyo dance. But maintaining this kind of semantic consistency is entirely up to you.

Variables

In the last section we saw that names can designate methods as well as values. While the importance of naming methods may be obvious to you by now, designating values by names is an even more important feature of programming. By naming values, we can create names that represent specific values, like the speed of a robot, or the average high temperature in the month of December on top of the Materhorn in Switzerland, or the current value of the Dow Jones Stock Index, or the name of your robot, etc. Names that designate values are also called *variables*.

Each variable in Java has three properties: a *name*, a *type*, and a *value*. The name and type will never change, but the value can (and usually will) change during the execution of a program, which is why they're called *variables*. A variable's type specifies the kind of values the variable can hold, and we'll look at available types shortly.

Java requires you to declare each of the variables you use in a program. "Declaring a variable" means telling Java the name and type of the variable. The following are examples of variable declarations:

```
int x;  
double averageTemperature;  
Scribbler robot;  
String myRobotName;
```

The above variable declarations declare four variables named `x`, `averageTemperature`, `robot`, and `myRobotName`. Variable `x` will hold an integer, `averageTemperature` will hold a double, `robot` will hold a Scribbler

object, and `myRobotName` will hold a string. For convenience you can declare several variables of the same type in one declaration. For example, the declaration

```
double a, b, c;
```

declares three variables of type `double`, named `a`, `b`, and `c`. This declaration is equivalent to

```
double a;  
double b;  
double c;
```

Types

Java has several built-in types of numeric data. The most common are:

- `int` – can hold a whole number (e.g., 5, -846, 0, 42, etc.). For reasons that are beyond the scope of this course, the smallest value that can be stored in an `int` is -2,147,483,648, and the largest value is 2,147,483,647.
- `double` – can hold a fractional number³ (e.g., -8.6, 598.09447, 0.0, etc.). `double` values can be much larger than `ints`.

Java also defines type `String` that holds textual data.

There are several other built-in types that will be discussed in later chapters of this course.

Finally, every class is also a type. So, for example, `Scribbler`, `dancingScribbler`, and `danceCompetition` are all valid Java types.

Values

³ Fractional numbers (i.e., those with a decimal point) are sometime called *floating point* numbers.

There are several ways you can assign a value to a variable, but the most common way is with the assignment statement. For example:

```
x = 45;
```

places the integer value 45 into `int` variable `x`. The value `x` held before executing this statement is replaced with 45. In other words, *a variable holds exactly one value at any given point in time; assigning a new value to a variable discards the previous value.*

The general description of an assignment statement is:

```
<variable_name> = <expression> ;
```

Assuming variables `x`, `averageTemperature`, `a`, `b`, `c`, and `myRobotName` are declared as in the previous section, the following statements are legal:

```
x = 29;
a = 56.5;
b = a + 984.2;
a = 947.28;
myRobotName = "Shrek";
```

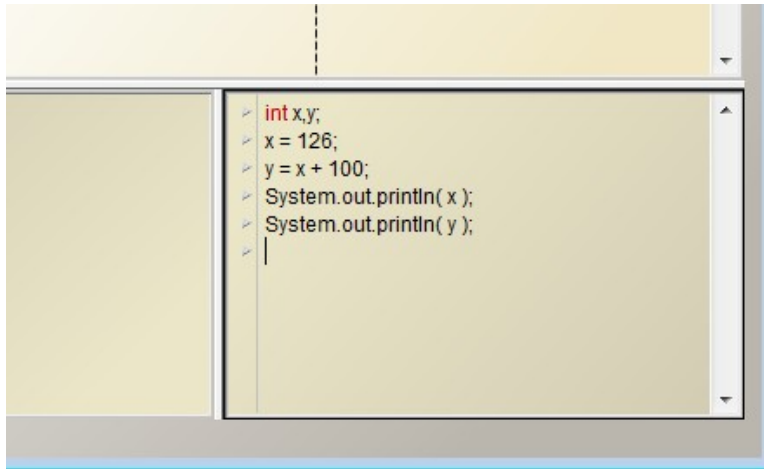
Do This: Open the `Chapter_3_Sandbox` project in BlueJ, and be sure the codepad is shown. The BlueJ codepad allows you to enter pretty much any Java statement, including variable declarations. Enter the following declarations into the codepad:

```
int x, y;
```

Now enter the following statements:

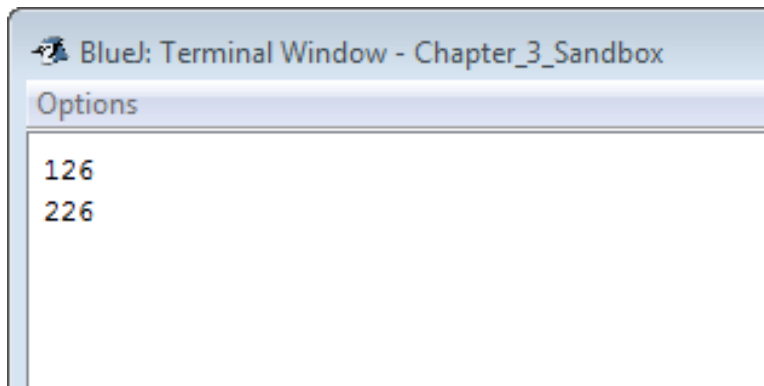
```
x = 126;
y = x + 100;
System.out.println( x );
System.out.println( y );
```

The codepad should look similar to:

A screenshot of a code editor window with a light beige background. The code is as follows:

```
> int x,y;  
> x = 126;  
> y = x + 100;  
> System.out.println( x );  
> System.out.println( y );  
> |
```

What happened when you executed the `System.out.println` statements? If everything worked properly you should have seen the values of variables `x` and `y` printed in the BlueJ Terminal Window:

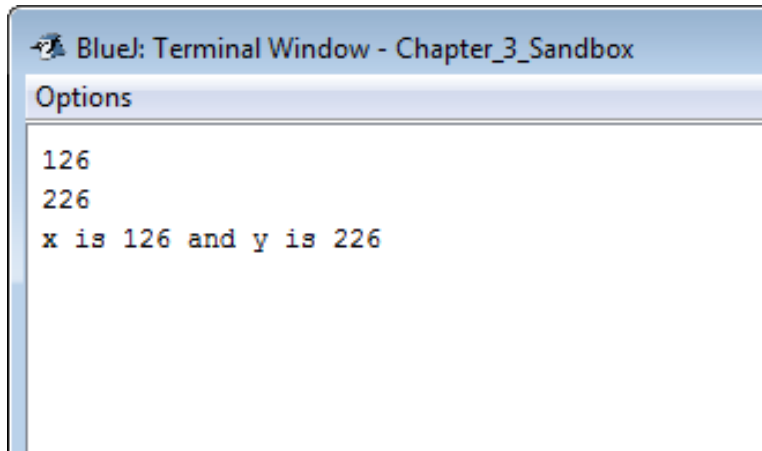
A screenshot of the BlueJ Terminal Window titled "BlueJ: Terminal Window - Chapter_3_Sandbox". It has an "Options" tab at the top. The terminal output shows:

```
126  
226
```

Now enter the following statements:

```
System.out.println("x is " + x + " and y is " + y );
```

What was printed in the BlueJ Terminal Window? It should look similar to:



```
BlueJ: Terminal Window - Chapter_3_Sandbox
Options
126
226
x is 126 and y is 226
```

Entering variable declarations, assignment statements, and `System.out.println` statements into the BlueJ codepad is an easy way to experiment with the behavior of assignment statements in Java. You are encouraged to make liberal use of this feature as you learn about expressions and assignment statement behavior.

There are some important things to understand about assignment statements:

1. When an assignment statement is executed, the expression on the right hand side of the statement is evaluated, and then that value is placed into the variable on the left hand side. For example, consider the following sequence of statements:

```
x = 45;
x = x + 10;
```

What value will be in variable `x` after the second statement is executed? If you answered 55 you are correct! The first statement places 45 into variable `x`. The second statement calculates the value of the expression `x + 10`, and because variable `x` contains 45 at this time, the result is 55. 55 is then placed into variable `x` (which replaces the 45 that was there). So, after the second statement is executed, variable `x` contains the value 55.

A very common action we'll want to perform in our programs is to increment the value of a variable. For example, if `int` variable `count` currently contains 10, we'd want it to contain 11 afterwards. Can you think of an assignment statement that will do this? Here's one way to do it:

```
count = count + 1;
```

Do you see why this works?

2. Although assignment statements look similar to algebraic statements, they are very different. For example, consider the following sequence of assignment statements (assuming variables `x` and `y` are both `ints`):

```
x = 15;  
y = x + 100;  
x = 20;
```

What values will be in variables `x` and `y` after the third statement is executed? `x` will be 20 and `y` will be 115. Did you think `y` should be 120? When the second statement is executed, `x` contains 15, so the result of `x + 100` (which is, not surprisingly, 115) will be placed into variable `y`. The third statement simply places 20 into variable `x`, leaving variable `y` alone.

It's important to realize that the second statement does *not* mean that variable `y` will always contain whatever is in `x + 100`, even if `x` changes. Instead it means that variable `y` will contain 100 + the value of variable `x` *right now*. If `x` changes value later on, it will not change the value of `y`.

3. Java requires that the type of the expression on the right hand side matches the type of the variable on the left hand side. Suppose variables `x` and `y` are `ints` and `a` and `b` are `doubles`. The following are legal assignment statements:

```
x = y + 45;
```

```
a = 148.3 - b;
```

However, the following are illegal:

```
x = 56.3;
y = a - 5.0;
y = 1.0;
```

Why are these illegal? Because the expressions on the right hand side are `doubles`, and the variables on the left hand side are `ints`. Java won't let us put `doubles` into `ints`.

Java will, however, let us put `ints` into `doubles`, so the following are legal:

```
a = x + 12;
b = 1;
```

int vs. double constants

Java is very picky about types so it's important to understand how `int` constants are different from `double` constants. `int` values are numbers without a decimal point, whereas `doubles` *must* have a decimal point. Thus, 42 is an `int`, while 42.0 is a `double`.

Do This: Enter the above declarations and assignments in the BlueJ codepad, and use `System.out.println` to verify the behavior of these statements. What happened when you entered the illegal expressions?

Arithmetic Expressions

The following arithmetic operators are defined in Java:

- + Addition
- Subtraction
- * Multiplication
- / Division
- % Modulus operator
- () Parentheses to override order of operation precedence

For example, the following assignment statements are all legal (assuming all variables are properly declared):

```
x = (a + b + c) / 3.0;  
y = x + 3 * z;
```

Operators for addition, subtraction, and multiplication do exactly what you'd expect; however, the behavior of the division operator needs some explanation, and what the heck is a modulus operator?

The division operator (`/`) behaves differently depending on the types of values it's working with. If both values are `ints`, it behaves as an *integer division*, but if one or both values are `doubles` it behaves as a *floating point division*.

Floating point division produces a `double` result and is probably what you expect division to do. For example, the statement

```
a = 23.0 / 4.0;
```

places the value 5.75 into variable `a` (assuming, of course, that `a` was declared as a `double` variable). This is probably not at all surprising to you.

Before proceeding with a discussion of integer division let's review division. When children first learn division they usually learn that division produces a *quotient* and a *remainder*. For example, 23 divided by 4 is 5 remainder 3; that is, the quotient is 5 and the remainder is 3. Does this bring back fond memories of elementary school? It should!

Integer division (i.e., when both values are `ints`) produces an `int` that is the quotient of the division. For example, the statement

```
x = 23 / 4;
```

places the value 5 into variable `x` (assuming `x` is declared as an `int`).

If integer division produces the quotient, is there a way to determine the remainder? Yes! This is what the modulus operator (`%`) does. For example, the statement

```
y = 23 % 4;
```

places the value 3 into variable `y`, because 23 divided by 4 is 5 remainder 3. The modulus operator is only defined for two `int` values; using a `double` as either value is illegal. So, for example, the following statement causes an error:

```
y = 23.0 % 4;
```

Do This: Assume that variables `a`, `b`, and `c` are declared as `double`, and variables `w`, `x`, `y`, and `z` are `ints`. What value is placed in the variables by the following assignment statements: (you can use a calculator if you wish)

```
x = 45;
y = x / 7;
z = x % 7;
a = 95.3;
b = a / 4.2;
c = x / 3.5;
w = x % 5;
```

Next, use the BlueJ codepad and `System.out.println` statements to see if your answers are correct.

Java Strings

A variable of type `String` (notice the capital S!) can hold a sequence of zero or more characters. Strings are useful to hold data such as someone's name, the name of a city or state, etc.

Consider the following Java statements:

```
String first, last, full;
```

```
first = "Alan";
last = "Turing";
full = first + " " + last;
```

Do This: Use the BlueJ codepad and `System.out.println` statements to determine what value each of the three `String` variables holds after executing the above assignment statements. (Note that there is a space between the two quotation marks in the last statement above.)

You probably realize that `String` constants consist of characters surrounded by double-quotation marks (`"`). Did you figure out what the `+` operator does when the values are `Strings`? It's called a *String concatenation* operator; it takes two `Strings` and produces a new `String` that contains the two `Strings` joined together.

Do This: Write an assignment statement similar to the last statement above that produces a full name in “last, first” order. For example, if `first` contains “Alan” and `last` contains “Turing”, `full` should contain “Turing, Alan”. Test your statement using the codepad and a `System.out.println` statement.

Do This: Using the BlueJ codepad, figure out what the `String` concatenation operator (i.e., `+`) does when one of the values is a `String` and the other is an `int` or a `double`. For example, you could try the following statements (assuming `x` is an `int` and `ans` is a `String`):

```
x = 42;
ans = "The answer is " + x;
```

A Calculating Program

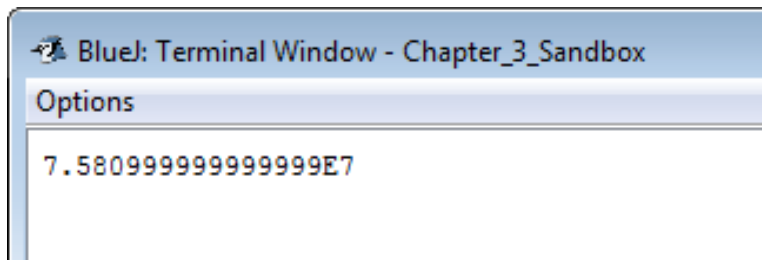
Ok, set your robot aside for just a few more minutes. You have now learned enough Java to write programs that perform simple, yet interesting, calculations. Here is a simple problem:

On January 1, 2008 the population of the world was estimated at approximately 6.650 billion people. It is predicted that at current rates of population growth, we will have over 9 billion people by the year 2050. A gross estimate of population growth puts the annual increase at +1.14% (it has been as high as +2.2% in the past). Given this data, can you estimate by how much the world's population will increase in this year (2008)? Also, by how much will it increase each day?

In order to answer the questions, all you have to do is compute 1.14% of 6.650 billion to get the increase in population this year. If you divide that number by 366 (the number of days in 2008) you will get the average daily increase. You can just use a calculator to do these simple calculations. You can also use BlueJ and Java to do this in two ways. First, you can use the BlueJ codepad as a calculator as shown below:

```
System.out.println( 6650000000.0 * 1.14 / 100.0 );
```

The terminal window shows the answer:



This answer may look a bit strange. When Java prints a large `double` value it prints it in scientific notation. The value `7.580999999999999E7` means $7.580999999999999 \times 10^7$, which is 75,809,999.9999999. So, in 2008 the population increased about 75.81 million.

Calculating the average daily increase can be done in the codepad with:

```
System.out.println( 75810000.0 / 366.0 );
```

The average daily increase is over 207 thousand people. So now you know the answers to our problem!

Just for fun, let's write a Java program to do the above calculations. A program to do the calculation is obviously going to be a bit of overkill. Why do all the extra work when we already know the answer? Small steps are needed to get to higher places. So let's indulge ourselves and see how you would write a Java program to do this. Below, we give you one version:

```
public class worldPop1
{
    public void main()
    {
        // declare population and growth rate variables and
        // set them to initial values
        double population = 665000000.0;
        double growthRate = 1.14 / 100.0;

        // declare variables for groths and caluclate
        // their values
        double growthInOneYear = population * growthRate;
        double growthInOneDay = growthInOneYear / 366;

        // print results
```

Limits of `int`

Java cannot represent the value 6.65 billion as an `int`, but it can represent this value as a `double`. Recall that `ints` must be between approximately -2 billion and +2 billion. When calculating the population in 2008 it was important that we used `6650000000.0` and not `6650000000!`

```

System.out.println( "World population on 1/1/2008 is "
    + population );
System.out.println( "On 1/1/2009 it will be "
    + growthInOneYear );
System.out.println( "The average daily increase is "
    + growthInOneDay );
    }
}

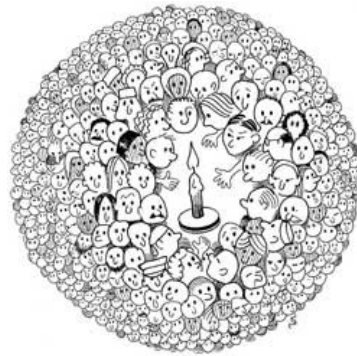
```

The program follows the same structure and conventions we discussed earlier. In this program we are not using the Myro/Java package so we don't need to have the `import` statement, nor do we need to declare a `Scribbler` variable. We have defined variables with names `population` and `growthRate` to designate the values given in the problem description. We've used a shorthand feature of Java that let's us declare a variable and assign a value to it in one statement. We also defined the variables `growthInOneYear` and `growthInOneDay` and used them to designate the results of the calculations. In the `main` method we first assign the values given, followed by performing the calculation. Finally, we print out the result of the computations.

Do This: Create class `worldPop1` in the `Chapter_3_Sandbox` project and edit it to contain the above code. Compile and execute it and observe the results.

Voila! You are now well on your way to learning the basic techniques in

The Energy Problem



The root cause of world energy problems is growing world population and energy consumption per capita.

How many people can the earth support? Most experts estimate the limit for long-term sustainability to be between 4 and 16 billion.

From: *Science, Policy & The Pursuit of Sustainability*, Edited by Bent, Orr, and Baker. Illus. by Shetter. Island Press, 2002.

computing! In this simple program we did not import anything, nor did we feel the need to define any methods (apart from `main`). This was a trivial program, but it should serve to convince you that writing programs to do computation is essentially the same as controlling a robot.

Using Input

The program we wrote above uses specific values of the world's population and the rate of growth. Thus, this program solves only one specific problem for the given values. What if we wanted to calculate the results for a different growth rate? Or even a different estimate of the population? What if we wanted to try out the program for varying quantities of both? Such a program would be much more useful and could be used over and over again. Notice that the program begins by assigning specific values to the two variables:

```
population = 6650000000.0;
growthRate = 1.14/100.0;
```

One thing you could do is simply modify those two lines to reflect the different values. However, typical programs are much more complicated than this one and it may require a number of different values for solving a problem. When programs get larger, it is not a good idea to modify them for every specific problem instance but it is desirable to make them more useful for all problem instances.

One way you can achieve this is to let the user tell our program what values he or she wants us to use. Myro/Java method `MyroGUI.inputDouble` is one way to do this:

```
import Myro.*;

/**
 * Program to calculate the population of the world, letting
 * the user input a current population and assumed growth
 * rate.
 *
 * @author Douglas Harms
```

```
* @version 20 April 2011
*/
public class worldPop2
{
    public void main()
    {
        // declare population and growth rate variables
        double population;
        double growthRate;

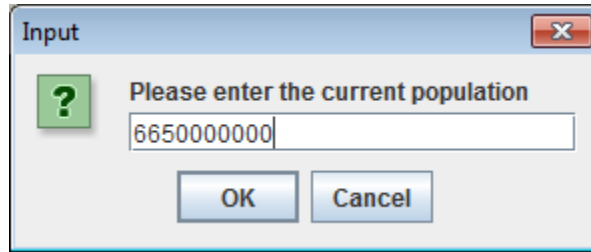
        // ask user for current population and growth rate
        population = MyroGUI.inputDouble
            ( "Please enter the current population" );
        growthRate = MyroGUI.inputDouble
            ( "Please enter the growth rate" ) / 100.0;

        // declare variables for growths and calculate
        // their values
        double growthInOneYear = population * growthRate;
        double growthInOneDay = growthInOneYear / 366;

        // print results
        System.out.println( "World population today is "
            + population );
        System.out.println( "In one year it will be "
            + growthInOneYear );
        System.out.println( "The average daily increase is "
            + growthInOneDay );
    }
}
```

Do This: Create class `worldPop2` in `Chapter_3_Sandbox`, compile and execute it. (Note: You do not need to place the parameter to `MyroGUI.inputDouble` on a separate line. We did in the above program only because the parameter wouldn't fit on one line in the textbook.) Execute the program several times, using different values for initial population and growth rate.

As you should have observed when you executed this program, method `MyroGUI.inputDouble` takes one parameter (a `String`) that is displayed in a dialog box when asking the user to input a value:



`MyroGUI.inputDouble` lets the user enter a double number (i.e., the number can have a decimal point) and returns that value to the program. The first time we invoke `MyroGUI.inputDouble` we ask the user to enter the current population and we place the value s/he enters into variable `population`. The second time we call `MyroGUI.inputDouble` we ask the user to enter a growth rate, and we take the value the user enters, divide it by 100.0, and place the result of the division into variable `growthRate`.

Myro/Java also defines methods `MyroGUI.inputInteger` and `MyroGUI.inputString`. These methods behave similarly to `MyroGUI.inputDouble`, except the value returned is an `int` or `String`, respectively.

Robot Brains

Writing programs to control your robot is therefore no different from writing a program to perform a computation. They both follow the same basic structure. The only difference is that all robot programs you will write will make use of the Myro/Java library. There will be several robot programs that will require you to obtain input from the user (see exercises below). You can then make use of the `MyroGUI.input` methods as described above.

One characteristic that will distinguish robot programs from those that just do computations is in the amount of time it will take to run a program. Typically, a program that only performs some computation will terminate as soon as the computation is completed. However, it will be the case that most of the time

your robot program will require the robot to perform some task over and over again. Here then, is an interesting question to ask:

Question How much time would it take for a vacuuming robot to vacuum a 16ft x 12ft room?

Seemingly trivial question but if you think about it a little more, you may reveal some deeper issues. If the room does not have any obstacles in it (i.e. an empty room), the robot may plan to vacuum the room by starting from one corner and then going along the entire length of the long wall, then turning around slightly away from the wall, and traveling to the other end. In this manner, it will ultimately reach the other side of the room in a systematic way and then it could stop when it reaches the last corner. This is similar to the way one would mow a flat oblong lawn, or even harvest a wheat field, or re-ice an ice hockey rink using a Zamboni machine. To answer the question posed above all you have to do is calculate the total distance travelled and the average speed of the vacuum robot and use the two to compute the estimated time it would take. However, what if the room has furniture and other objects in it?

You might try and modify the approach for vacuuming outlined above but then there would be no guarantee that the floor would be completely vacuumed. You might be tempted to redesign the vacuuming strategy to allow for random movements and then estimate (based on average speed of the robot) that after some generous amount of time, you can be assured that the room would be completely cleaned. It is well known (and we will see this more formally in a later chapter) that random movements over a long period of time do end up providing uniform and almost complete coverage. Inherently this also implies that the same spot may very well end up being vacuumed several times (which is not necessarily a bad thing!). This is similar to the thinking that a herd of sheep, if left grazing on a hill, will result, after a period of time, in a nearly uniform grass height (think of the beautiful hills in Wales).

On the more practical side, iRobot's Roomba robot uses a more advanced strategy (though it is time based) to ensure that it provides complete coverage. A more interesting (and important) question one could ask would be:

Question: How does a vacuuming robot know that it is done cleaning the room?

Most robots are programmed to either detect certain terminating situations or are run based on time. For example, run around for 60 minutes and then stop. Detecting situations is a little difficult and we will return to that in the next chapter.

So far, you have programmed very simple robot behaviors. Each behavior which is defined by a method, when invoked, makes the robot do something for a fixed amount of time. For example, the `yoyo` behavior from the last chapter, when invoked as:

```
yoyo( 0.5, 1 );
```

would cause the robot to do something for about 2 seconds (1 second to go forward and then 1 second to move backward). In general, the time spent carrying out the `yoyo` behavior will depend upon the value of the second parameter supplied to the function. Thus if the invocation was:

```
yoyo( 0.5, 5.5 );
```

the robot would move for a total of 11 seconds. Similarly, the `dance1` behavior (in the `danceCompetition` class defined in the previous chapter) will last a total of eight seconds. Thus, the total behavior of a robot is directly dependent upon the time it would take to execute all the commands that make up the behavior. Knowing how long a behavior will take can help in pre-programming the total amount of time the overall behavior could last. For example, if you wanted the robot to perform the dance moves for 60 seconds, you can repeat the `dance1` behavior ten times. You can do this by simply invoking the `dance1` method 10 times. But that gets tedious for us to have to repeat the same method invocation so many times. Computers are designed to

do repetitious tasks. In fact, repetition is one of the key concepts in computing and all programming languages, including Java, provide simple ways to specify repetitions of all kinds.

Definite Repetition in Java – the `for` Loop

If you wanted to repeat the `dance1` behavior at $\frac{3}{4}$ speed 10 times in a program, all you have to do is:

```
for( int i=0; i<10; i++ )
{
    robot.dance1( 0.75 );
}
```

This is a new statement in Java: the `for`-statement. It is also called a *definite loop statement* or simply a *for-loop*. It is a way of repeating something a definite (i.e., fixed) number of times. The basic syntax of a `for`-loop in Java is:

```
for ( int <var> = 0; <var> < <numTimes>; <var>++ )
{
    <do something>
    <do something>
    ...
}
```

The `for`-loop specification begins with the word `for`, which is followed by the loop specification enclosed in parentheses, which is followed by the body of the loop enclosed in curly braces. Let's look at each of these parts separately.

The loop specification consists of three items separated by semicolons (`;`) and enclosed in parentheses. The first item consists of an `int` variable declaration that also sets the initial value to 0 (e.g., `int i=0`); the variable is called the *loop control variable*. The second item consists of a comparison between the loop control variable and the number of times the loop should repeat (e.g., `i<10`). The last item increments the loop control variable by one, usually using the `++` operator (e.g., `i++`).

The name of the loop control variable is not important right now, so you can use any variable name you like. For example, the for-loop specification:

```
( int count=0; count<42; count++ )
```

specifies a loop control variable named `count` that will repeat the loop 42 times.

The body of the loop consists of various Java statements, all enclosed in curly braces (`{ }`). These are the statements that will be executed the number of times specified in the for-loop specification.

Do This: Let's try this out on the robot by modifying the `competition_two` program from earlier in this chapter to have the robot perform the first dance routine 10 times.

```
import Myro.*;

/**
 * This program has the robot perform the first dance routine
 * at half speed 10 times.
 *
 * @author Douglas Harms
 * @version 16 April 2011
 */
public class competition_three
{
    // declare the danceCompetition object
    private danceCompetition robot;

    public void main()
    {
        // create a danceCompetition object and connect it
        robot = new danceCompetition();
        robot.connect( "COM3" );

        // have the robot dance 10 times
        System.out.println( "About to dance..." );
        for( int danceSteps=0; danceSteps<10; danceSteps++)
        {
```

```

        robot.dance1( 0.5 );
    }
    System.out.println( "...Done dancing!" );

    // disconnect from the robot
    robot.close();
}
}

```

Notice that we have used `danceStep` (a more meaningful name than `i`) as the loop control variable. When you run this program, the robot should perform the dance routine ten times.

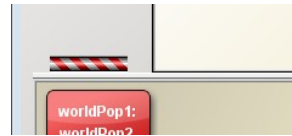
Do This: Modify the value specified in the for loop specification to have your robot repeat the dance routine various number of times. **Caution:** keep in mind that the first dance routine takes about 8 seconds to perform, so specifying a large number of times to perform this routine may take a long time. For example, repeating the routine 100 times will take over 13 minutes to complete!

Do This: Modify the program so that the user is asked how many times to repeat the dance routine. Hint: declare an `int` variable named something like `repeatCount`. Use `MyroGUI.inputInteger` to get a value from the user, then place this value into variable `repeatCount`. Use `repeatCount` in the loop specification as follows:

```
( int danceSteps=0; danceSteps<repeatCount; danceSteps++ )
```

Aborting a Java Program

There are times when something goes wrong and you need to abort a running Java program. To do this, right-click on the horizontal “barber-pole” icon in the BlueJ window and select “Reset Java Virtual Machine”. Your program will stop.



Indefinite Repetition in Java – the `while` Loop

The for-loop is *the* loop to use when it's known exactly how many times something needs to repeat (e.g., 10, 42, `repeatCount`, etc.). However, there are times when you want to keep repeating something until a particular condition occurs, rather than repeating for a specific number of times. For example, you might want to repeat something for a particular length of time, or until your robot hits a wall, or until a light goes on, etc. These types of loops are called *indefinite loops* because they execute an indefinite (i.e., not fixed) number of times.

The `while` statement in Java is used for indefinite loops. There are several ways we can use a while statement, and we'll discuss these in later chapters. Here is the way a while loop can be used to execute a block of code for a particular amount of time:

```
while( MyroUtils.timeRemaining( <numSeconds> ) )
{
    <do something>
    <do something>
    ...
}
```

For example, the following Java code will instruct your robot to perform the first dance routine at half speed for 15 seconds:

```
while( MyroUtils.timeRemaining( 15 ) )
{
    robot.dance1( 0.5 );
}
```

Do This: Create a Java program named `competition_four` that is identical to program `competition_three` discussed above, except that it instructs your robot to perform `dance1` for 10 seconds instead of 10 times. Next, modify this program so it asks the user for the speed to dance and the amount of time to dance.

You may be wondering about what happens if the time “runs out” during the middle of a dance routine – does the robot stop in the middle or does it keep

going to finish the routine? This is an excellent question! Can you find a way to test this, perhaps using the program you just wrote? One way is to have your robot perform the dance routine for 1 second. Recall that it takes about 8 seconds to perform the dance routine, so if the robot stops after completing one eighth of the routine, then you know that Java cuts off the routine as soon as the time's up; if, on the other hand, the robot completes the entire routine, you know that Java doesn't stop it in the middle⁴.

Summary

This chapter introduced the basic structure of Java (and robot) programs. We also learned about *names* and *values* in Java. Names can be used to designate classes, methods, and values. The latter are also called *variables*. Java provides several different types of values: integers (`int`), floating point numbers (`double`), and strings (`String`). Most values have built-in operations (like addition, subtraction, etc.) that perform calculations on them. Myro/Java provides simple facilities for obtaining input from the user. All of these enable us to write not only robot programs but also programs that perform any kind of computation. Repetition is a central and perhaps the most useful concept in computing. In Java you can specify repetition using either a `for`-loop or a `while`-loop. The latter are useful in writing general robot brain programs. In later chapters, we will learn how to write more sophisticated robot behaviors.

Myro/Java Review

```
MyroUtils.timeRemaining(<seconds>)
```

This is used to specify timed repetitions in a `while`-loop (see below).

```
MyroGUI.inputInteger( <message> )
```

```
MyroGUI.inputDouble( <message> )
```

```
MyroGUI.inputString( <message> )
```

These methods display a dialog box containing the `String` `<message>`, the

⁴ In fact, Java does not stop in the middle of a routine. The `while` loop presented here means "if the time is not up yet, execute the body of the loop; when that's done, check the time again, and if it's not up yet, execute the body again; keep doing this until the time is up."

allows the user to enter a value that must be an `int`, `double`, or `String`, depending on the method. The value entered by the user is returned to the program.

Java Review

Values

Values in Java can be numbers (integers or floating point numbers) or strings. Each type of value can be used in an expression by itself or using a combination of operations defined for that type (for example, `+`, `-`, `*`, `/`, `%` for numbers). Strings are considered sequences of characters (or letters).

Names

A name in Java must begin with either an alphabetic letter (`a-z` or `A-Z`) or the underscore (i.e. `_`) and can be followed by zero or more letters, digits, or underscore characters.

Basic Structure of a Myro/Java Program

```
import Myro.*;

/**
 * Write a description of class className here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class className
{
    // declare the Scribbler object
    private Scribbler robot;

    public void main()
    {
    }
}
```

Text Output Statement

```
System.out.println( <String expression> );
```

Prints out the result of the expression on the screen (in the BlueJ Terminal Window). When no expression is specified, it prints out an empty line

Variable Declaration

```
<variable type> <variable name>;
```

This declares a variable named <variable name> to be of type <variable type>. Possible types are `int`, `double`, `String`, and any class name (e.g, `Scribbler`).

Assignment Statement

```
<variable name> = <expression> ;
```

This is how Java assigns values to variables. The value generated by <expression> will become the new value of <variable name>. Java requires the type of the variable and expression to match.

Repetition

```
for( int <var>=0; <var> < <limit>; <var>++ )  
{  
    <do something>  
    <do something>  
    ...  
}
```

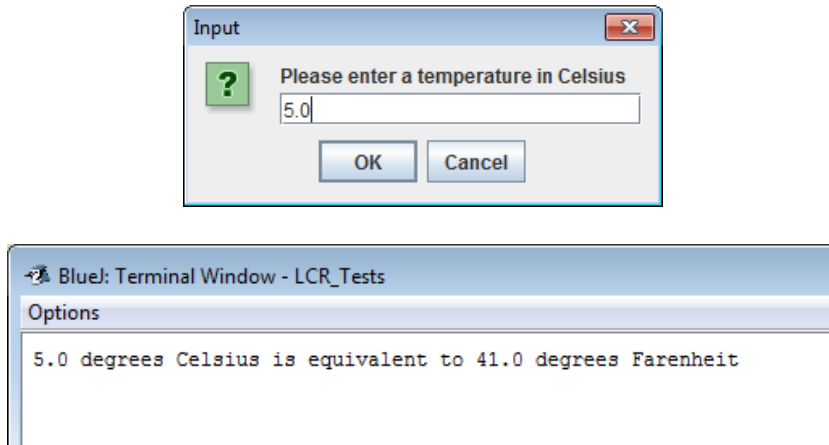
```
while( MyroUtils.timeRemaining(<seconds> ) )  
{  
    <do something>  
    <do something>  
    ...  
}
```

These are different ways of doing repetition in Java. The first version (called a definite loop) executes the body a fixed number of times. The second version

(called an indefinite loop) will carry out the body for `<seconds>` amount of time. `MyroUtils.timeRemaining` is a Myro/Java method (see above).

Exercises

1. Write a Java program to convert a temperature from degrees Celsius to degrees Fahrenheit. Here is a sample interaction with such a program:



The formula to convert a temperature between Celsius and Fahrenheit is: $C/5 = (F - 32) / 9$, where C is the temperature in degrees Celsius and F is the temperature in degrees Fahrenheit.

2. Write a Java program to convert a temperature from degrees Fahrenheit to degrees Celsius.

3. Write a program to convert a given amount of money in US dollars to an equivalent amount in Euros. Look up the current exchange rate on the web (see xe.com, for example).

4. Modify the version of the dance program above that uses a `for`-loop to use the following loop:

```
for danceStep in [1,2,3]:  
    dance()
```

That is, you can actually use a list itself to specify the repetition (and successive values the loop variable will take). Try it again with the lists [3, 2, 6], or [3.5, 7.2, 1.45], or ["United", "States", "of", "America"]. Also try replacing the list above with the string "ABC". Remember, strings are also sequences in Python. We will learn more about lists later.

5. Run the world population program (any version from the chapter) and when it prompts for input, try entering the following and observe the behavior of the program. Also, given what you have learned in this chapter, try and explain the resulting behavior.

6. Use the values 9000000000, and 1.42 as input values as above. Except, when it asks for various values, enter them in any order. What happens?

7. Using the same values as above, instead of entering the value, say 9000000000, enter 6000000000+3000000000, or 450000000*2, etc. Do you notice any differences in output?

8. For any of the values to be input, replace them with a string. For instance enter "Al Gore" when it prompts you for a number. What happens?

9. Rewrite your solution to Exercise 4 from the previous chapter to use the program structure described above.

10. You were introduced to the rules of naming in Python. You may have noticed that we have made extensive use of *mixed case* in naming some entities. For example, `waitTime`. There are several naming conventions used by programmers and that has led to an interesting culture in of itself. Look up the phrase *CamelCase controversy* in your favorite search engine to learn about naming conventions. For an interesting article on this, see The Semicolon Wars (www.americanscientist.org/issues/pub/the-semicolon-wars).

11. Experiment with the `speak` function introduced in this chapter. Try giving it a number to speak (try both integers and floating point numbers). What is the largest integer value that it can speak? What happens when this limit is exceeded? Try to give the `speak` function a list of numbers, or strings, or both.

12. Write a Python program that sings the ABC song: *ABCD...XYZ. Now I know my ABC's. Next time won't you sing with me?*

Sidebar Ideas:

indenting?

Why two types of numeric data?