

Relationship of class to object

Writing and programming

Writing a program is similar to many other kinds of writing. The purpose of any kind of writing is to take your thoughts and let other people see them. And you have to follow the rules of writing for your paper to mean anything. For example, when you write a term paper, you have to follow rules for punctuation, paragraphs and footnotes. If you don't follow the rules, the teacher may reject your paper.

Before we start describing program writing, we take a side trip. Poetry is a form of writing that tends to have more rules than prose or narratives. One form of poetry that most school children learn is Haiku, an ancient Japanese poetry style. The rules for Haiku are very simple but very strict. A Haiku poem is always three lines long and the lines have, in order, 5, 7 and 5 syllables. Take a moment and write some Haiku poems about various parts of robots. Here are a couple of examples:

Little motor runs
Backward, forward, on and off
Makes my robot move

Sensor on my bot
How I wonder what thou art
Touch, light, infrared?

As you can see, the poems are not literary gems. But they do convey an idea of the nature of motors and sensors. Obviously, a computer is not going to accept a Haiku poem as a program. But the process of writing a Haiku poem and a program are similar. You express your ideas in a very structured writing style. As a further exercise, write a limerick about robots.¹

The following steps introduce you to the tools you will be using. There are two distinct things you will have to learn in order to be a Java programmer: development tools and the Java language. The next sections are mostly about the tools. You will be asked to cut and paste code into the IDE editor, run the compiler and look at the output. The hope is that in a very short time, using the tools will

¹ Limericks are traditional Irish poems. They consist of five lines, with the first, second and fifth lines rhyming and the third and fourth lines rhyming. Limericks also tend to be somewhat bawdy.

become automatic. When you start your own programs, you want to concentrate on your code and not be worried about how to invoke the compiler. Writers need to concentrate on what they are trying to say, not on how to use a word processor or whether they need to use a colon or semi-colon. As a programmer, you need to be so comfortable with your development environment that you don't consciously think about compiling code, you just do it. To get to that point, you need practice and that is next.

Step 1 – Empty.java

```
// a very simple class
class Empty {
}
```

You now should have written a Haiku about robots. Now we start writing computer programs. Just as you had to follow the style requirements for Haiku, you also have to follow the style requirements for Java. At left is the simplest object possible in Java. It is completely empty. It has no properties and no methods. But it does have the three essential parts of a Java class: the declaration word “class”, a name, in this case “Empty”, and curly braces, “{” and “}”, that show the beginning and end of the class declaration. Cut and paste this class into a new file in the IDE. When you create the new file, name it “Empty”. Then compile it using the JDK compiler. What you should get is the message “process complete”.

Next, you should experiment a little. Try taking out one of the curly braces. Or, misspell class. Or, think of some other way to screw it up. Each time, the compiler will generate error messages, *compile-time errors*. Most error messages have line numbers to tell you which line is incorrect. Always check the line in the error message first. If you cannot find the error, look above the line number listed. Always remember to read what you wrote, not what you intended to write. One technique I have often used is explaining the code to my wife. She is not a programmer and doesn't understand much of what I am saying. But saying it forces me to read what I wrote. And, frequently, that is enough for me to find my error.

Become good friends with the error messages. If you start seriously programming you will see these guys a lot.

Finally, try executing Empty. What do you expect will happen?

As it turns out, you generate a *run-time error*. Why? That's for the next section.

Step 2 – NotSoEmpty.java

```
// a very simple program
class NotSoEmpty {
    public static void main(String[] arg) {
    }
}
```

In the last section, we introduced Empty. This section introduces the class NotSoEmpty. NotSoEmpty is not only a class but it is also a *program*. In Java, every class can be a program but does not have to be. To be a program, a class needs an *entry point*. An entry point is simply the place to start. The designers of

Java decided the entry point should be a class method with the name of “main”. The designers made the decision to call it main because they could. As you get into programming you will find many examples of “Because I said so.” In almost all cases, some thought went into the decision. You will be faced with similar decisions to make. When you need to make a decision, make it and move. Sometimes that means that you will need to throw away code because you made the wrong decision. Don’t sweat it when it happens, just learn from your mistake and move on. The only way for you to learn how to make good choices is to practice making choices.

Copy and paste NotSoEmpty into your IDE and run it. What do you expect will happen when you run NotSoEmpty?

As it turns out, it seems that nothing happened. But, in fact, something did happen. The program successfully ran without an error. It didn’t do anything, but it also did not do nothing!

Step 3 – MotorSubset.java - class

The next few pages will explore the Motor class from LeJOS. Previously, we looked at the UML class diagram for Motor. Now, we will start looking at the actual code. We begin with a subset of Motor that we call “MotorSubset”. You should cut and paste this code into JCreator and compile it using lejos, which is tool 2 and can be invoked with control-2.

```
1 import josx.platform.rcx.ROM;
2
3 public class MotorSubset
4 {
5     private char iId = 'A';
6     private short iMode = 4;
7     private short iPower = 3;
8
9     public static final int FORWARD = 1;
10
11     public static final MotorSubset A = new MotorSubset ();
12
13     private MotorSubset() {}
14
15     public final void forward()
16     {
17         iMode = FORWARD;
18         controlMotor (iId, FORWARD, iPower);
19     }
20
21     public static void controlMotor (char aMotor, short aMode,
22                                     short aPower)
23     {
24         ROM.call (6734, 8192 + aMotor - 'A', aMode, aPower);
25     }
26 }
```

Line 1 is an import statement. Import statements must be placed before the class statement. Import statements are used to gain access to other classes or objects. In this case, we import the class ROM. ROM has a static method, controlMotor, which allows us to control the motor in a very detailed, hardware-oriented way.

Line 3 is our class statement. We create a new class and name it MotorSubset. We gave it that name because it is a subset of the Motor class. There is no requirement that we have to name it that but it makes sense to do so. When you program, the names you give things should always be descriptive of what the thing is. Line 4, an open curly, {, starts the *definition* of the class MotorSubset.

Lines 5-7 define properties of MotorSubset. The first one, iId, is the identification, or ID, property for this instance of motor. It is defined to be of *type char* or character integer. In this case, it is initialized to the letter “A”. The next two properties are iMode and iPower. iMode is the mode of this

instance of MotorSubset. Physical motors can be in one of four modes: forward, reverse, stopped (no power, brake on) or float (no power, brake off). iPower is the power level applied to the motor. It ranges from 0, no power, to 7, full power. Both iMode and iPower are of type *short* or short integer. A short integer variable contains an integer with a range of –32,768 to 32,767. For most robotic applications, this range is quite sufficient. Both char and short are *primitives*. There are 8 primitive types in Java, which will be described in a [later section](#).

Line 9 defines a variable FORWARD of type *int*. The keyword *final* makes it illegal to change the value of FORWARD; in class MotorSubset, FORWARD will always have the value 1. Any place that you would want to use an integer with a value of 1, you can use FORWARD instead. In the method forward, below, you will see why this is useful.

Line 11 declares a property or variable, A, of type MotorSubset. As opposed to lines 5-9, which declare primitive properties, Line 11 declares a [reference](#) property. Reference properties are used to hold references to objects. Objects must be created before they can be used. That is what the *new* operator does; it creates a new instance of a MotorSubset object. We then place a pointer to the newly created MotorSubset object into the variable A. This is a class or static property because there is only one motor attached to the RCX on port A. We don't want to create more than one object in our programs that refer to the physical A motor on our robot.

Lines 15-19 define the forward method. The open curly, {, on line 15 starts the definition of forward and the close curly, }, on line 19 ends it. forward is an instance method; we call it in the context of a particular motor. Since A is the only motor that we have defined, we will have to call it as A.forward(). An example in the next section will make this clear. In line 17, we set the property iMode to FORWARD. To the compiler, this is exactly the same as assigning the value 1 to iMode. But which is clearer to you, or the person you give your code to, 1 or FORWARD? Using constants can make your code much more readable and maintainable. So use constants when you need them to make your code more clear. In line 18, we call controlMotor to tell the physical motor to turn on in the forward direction. This emphasizes why iMode is a private property. Just changing iMode does not have an effect on the physical motor. The motor's state changes by calling the method ROM.call. As you may have guessed, ROM is just another class like MotorSubset and call is a static method in ROM.

Lines 21-25 define the method controlMotor. controlMotor takes three parameters: the motor ID, the mode to put the specified motor into, and the power level. The only thing controlMotor does is to make a minor adjustment to the parameters and pass them onto the method that actually controls the RCX.

Finally, line 26 is another close curly, }. This ends the declaration of the class MotorSubset.

Copy and paste this code into a file named MotorSubset (make sure that you don't copy the line numbers into your source file). Then compile it using lejos, tool 2.

Now, it is time for a stretch. Look back to the [UML diagram](#) of Motor. There are three other methods similar to forward: backward, stop (stop, brake) and flt (stop, no brake). The mode parameters are backward 2, stop 3 and float 4. Implement these three new methods.

Step 4 – MotorSubset.java – program

In Java, a program is a class with a main method. This main method is very simple. It calls the forward method of the static instance A of the MotorSubset class. Copy and paste this into your MotorSubset.java file. Make sure that it is inside the class definition and outside of any other method definitions. Compile the file, lejos, tool 2, and download it to your robot, lejos, tool 1. You may need to download the firmware, firmwaredl, tool 3, before you can download your program.

1	
2	public static void main(String[] args)
3	{
4	Motor.A.forward();
5	wait();
	}

Once you have successfully downloaded the program, run it. What happens?

Replace the forward() method with the backward method that you created in the last step. Compile, download and run it. Does it do what you expect?

1	import josx.platform.rcx.Motor;
2	
3	public class RoverMotors
4	{
5	
6	public static void main(String[] args)
7	{
8	Motor.A.forward();
9	Motor.C.forward();
10	try {
11	java.lang.Thread.sleep(1000);
12	} catch (java.lang.InterruptedException e) {
13	}
14	Motor.A.backward();
15	Motor.C.backward();
16	try {
17	java.lang.Thread.sleep(1000);
18	} catch (java.lang.InterruptedException e) {
19	}
20	}
21	}
22	
23	
24	
25	
26	

Step 5 – RoverMotors.java

RoverMotors

Name

Blocks – scope – “{“ & “}”

Attributes

Methods

Private vs. public

Static vs. instance

Constructors

Composition

.equals()

.clones()

