9 Moving Beyond Robots to Objects

In this Chapter we discuss object-oriented programming and give some additional classes that you can use in robot programs.

9.1 Objects

I'm going to tell you a story to help you think about objects and how we deal with them.

Objects are electronic things.

They are active things. They can do stuff and they can remember stuff. They can talk to other objects.

Therefore they are more like people and cats than like rocks and trees.

You will probably find it advantageous to think of objects as if they were real rather than electronic. Thinking of them like people is a pretty good metaphor.

Just like people they are active. Just like people we can make requests of them. Just like people they themselves control what they do.

Most objects, though they can be active, are pretty passive. They are only active when they are asked to do some specific thing.

Mostly what objects do is provide services to other objects. When one object is active, it might ask another object to perform a service--maybe asking it for a piece of information or to do some task.

When an object asks another for a service, the asker is taking a "client" role and the askee is taking the "server" role.

When a client asks a server to perform a service it waits (politely) until the server completes the request and then it resumes its own activity. This is most helpful when the server needs to return information to the client. It is also possible to arrange things so the client continues its activity instead of waiting, but this is less usual.

There are many kinds of objects in a "system". What kind of object a thing is determines what services it can provide. If you ask an object to do something that isn't appropriate, that is an error. In Java, the computer won't let that happen.

So far in the robot programming world, the primary objects that we have seen are the various kinds of robots. We ask robots to do things (perform services) and they remember things (that they have beepers or not). Street corners are almost like objects, but in the Java program that runs our robot programs they are a bit simpler. However, we can implicitly ask a street corner if it has any beepers. Beepers aren't objects at all, since we don't ask them for any services. They are much more primitive. We have also seen Strategy objects and a few others. These do have behavior.

In the previous chapter we did see one other kind of object, as well. This was the die that was rolled to determine how long a Philosopher would eat or think.

For the most part in what we have done, the main task block has been our only client and the robots have been servers. However, sometimes we have had one robot ask another to do something. Then the first robot becomes a client of the second, which itself becomes a server for the first. A client server transaction is always initiated by the client. The server may return information to the client when performing the service. For example if robot Tony asks robot Lisa if it has anyBeepersInBeeperBag, Lisa will return a boolean value. Other services (void methods) don't return such information. Likewise, our robots delegated actions to strategies they were using.

Since objects, including robots, are things, they have names. We saw in Chapter 7 that we can have aliases for robots. The same is true for objects in general. The names we have been giving robots are called variables. This is because the object that they refer to can change as the program progresses. A variable of robot type, or in general, object type, can refer to any object of its type or any subtype. This means that a variable of type ur_Robot can actually refer to a Racer robot. You should carefully distinguish between the variable that refers to a robot and the robot itself. A variable is just a name or "handle" that we use to get access to a robot so that we can ask it to perform services. We saw an example in Chapter 8 in which we didn't need to send any messages to a robot so we didn't use any variable to refer to it. This was only possible since the robot ran in its own thread and executed its own run method.

In Java, variables can refer to other things besides objects. Some of these variables are not references to things, but hold the things themselves. For example, Java has a type called int (short for integer) in which we can store integer (whole number) values.

```
int size = 20;
```

Note that we created this integer and gave it a value without using new. A variable that isn't a reference to an object, such as size, can only hold a value of its own precise type. We can perform arithmetic on int variables as you might hope.

```
int bigger = size + 5;
```

The new variable bigger will have value 25.

While Java does have a few things that are not objects, the interesting things, and the ones you can build are all objects. Also, these simpler things are always defined within classes in Java and are therefore related to objects, even if they aren't objects themselves.

9.2 Classes

In object-oriented languages like Java and Karel J. Robot, classes are used to describe objects. A class describes a set of objects of the same kind: Racers or Sweepers, for example. A class describes what an object can do (its services) in its methods. A class can also describe what an object can remember, though we haven't seen a lot of this yet.

So far we can refer to a robot using a variable, but it doesn't remember any name for itself. We can correct this as follows.

```
class BrainyRobot extends Robot
{     public BrainyRobot (String name, int street, int avenue,
Direction direction, int beepers)
      {     super(street, avenue, direction, beepers);
```

```
            myName = name; // Remember the name.
        }

        public String name() { return myName; }

        private String myName = null; //A variable in which to remember
the name.

}
```

The constructor for this class has an additional parameter of type String. A String is a sequence of Characters, usually written in double quotes. "Karel J. Robot" is a String. String is a Java class, so it describes objects of type String. When we create a BrainyRobot we must give it a name. The name of the robot doesn't need to be the same as the name of the variable that we use to refer to it, however.

```
BrainyRobot kjr = new BrainyRobot("Karel", 1, 1, North, 0);
```

The name is something that the robot remembers using its instance variable called myName. In the constructor there is a statement that gives this variable a value, namely the value of the name parameter. The robot will remember this value and we can ask any BrainyRobot for its name using the name method.

In Java we can print out a string so that the person running the program can see it by sending a println message to a special object called System.out. For example we could write out the name of the BrainyRobot referred to by kjr with:

```
System.out.println(kjr.name());
```

Here the System.out object is our server and we need to send it information about how to carry out its service. We send it a String object that we get from the robot kjr. Note that kjr is also a server here and we are asking it to perform its name service, which it does by returning the current value of its myName instance variable.

A good way to think about a class is that it is a factory for creating objects of that type. This is why we used the factory metaphor (Karel-Werke) in Chapter 1. Each instance that it creates has all of the methods and all of the instance variables within the object (robot) itself. The new operator is like a message to the factory to create an object, though the usual message (".") syntax isn't used for this. The additional information that we give when we create a new robot, such as the street and avenue numbers, are called parameters. We will learn more about parameters in this Chapter.

9.3 Java Strings

"Jane Smith" is a special kind of object, known to Java, called a String. A String is a sequence of characters. Like any other object, a String object performs services for us. The most important service is to remember the sequence of characters that make it up.

A String can tell us its length.

```
int val = someWords.length();
```

A String can also return a new string to us that has all of its own characters together with those of another string.

```
String more = someWords.concat("  And women's, too. ");
```

Distinguish carefully between the variable (a name) and the object to which it refers. The name is not an object. It refers to an object.

The String class describes more than 30 services that a String can provide. There is no service, however, to change the characters in a String object. Strings are immutable. However, a String variable can refer to different strings at different times.

```
public class String
{     ... // stuff left out
      public int length()
      {     ...
      }

      public String concat (String s)
      {     ...
      }
      ...
}
```

We can look at individual characters in a string if we know which index in the string to look at. Each character in the string is stored in a numbered slot, starting with zero. Therefore aString.charAt(5) represents the sixth character in the string, assuming it has length at least six. If not this is an error.

There is a shorthand in Java for string concatenation. It is just the plus sign: +. We could have written the concat example above more simply as:

```
String more = someWords +"  And women's, too. ";
```

The same operator, +, is also used to add numeric values as you might expect.

We can print out an informative message about a BrainyRobot with something like:

```
String message = "Hello, my name is " + kjr.name();
if (kjr.anyBeepersInBeeperBag())
{     message = message + " some ";
}
else
{     message = message + " no ";
}
```

```
message = message + " beepers in my beeper bag.";
System.out.println(message);
```

This might produce something like the following:

Hello, my name is Karel. And I have some beepers in my beeper bag.

As we said above, in Java, not everything is an Object. Numeric values like 5 aren't objects, since they don't have behavior and can't remember things. They just are. (Like rocks.) All of the complex things that a programmer can define are objects, however. Java also has a large number of classes (like String) that come with the system, so you have lots of possibilities for using objects without building any classes. But usually, you want to build your own kind of objects to do interesting things.

Note that in Java, Strings are objects, but they don't behave polymorphically. This is because the class they are defined in is declared "final." This prohibits you from building a subclass of String. This has both advantages and disadvantages. The advantage is that you can always depend on how a string will behave. See Problem 31 of Chapter 6. The disadvantage is that you have to live with the behavior defined in the library. Methods can also be final. This means that they cannot be overridden. A final class implies that all of its methods are final.

9.4 Parameters for Robot Methods

So far, all of our constructors have had parameters, but few of our methods has. Often, when we ask an object to perform a service, we need to send along additional information. This can be data of any type, including other objects.

Suppose we are building the Mason class of Chapter 4. It might be better to be able to tell the Mason how high to build the wall.

```
public void buildWall(int height)
{     for(int i = 0; i < height; ++i)
      {     putBeeper();
            move();
      }
}
```

Inside the method, the robot will be able to use the parameter as an ordinary variable. But the parameter is defined only within the single method itself. Instance variables on the other hand are visible throughout the object: in every method.

Now, if we have a Mason named Ken we can say

```
Ken.buildWall(8);
```

And if Ken has enough beepers and is in the right place when we send the message, we will get our wall eight beepers high.

Here is a task that would be difficult without parameters. Suppose that we have one robot inside a rectangular room. We could ask that robot to measure the room, determining its area. Suppose that we have another robot outside the room that is supposed to put down a number of beepers equal to the area of

the room. We can have the second robot ask the first how big the room is, but to do this the second robot needs to know who to talk to. We have solved this kind of program before by defining the first robot inside the second, or by having the robots meet on the same corner, but there is another way.

```
class RoomMeasurer extends Robot
{     public void measureRoom(){…}
      public int sizeOfRoom(){…}
      private int size = 0;
}
```

This is the general form for the room measuring robot. Notice that we have given the size instance variable an initial value of zero, in case someone asks for the size of the room before measuring it.

```
class BeeperPutter extends Robot
{     public void learnCollaborator(RoomMeasurer r)
      {     mycollaborator = r;
      }
      public void putBeepers()
      {     if(myCollaborator != null)
            {     int size = myCollaborator.sizeOfRoom();
                  for (int i = 0; i < size; ++i)
                  {     putBeeper();
                  }
            }
      }
      private RoomMeasurer myCollaborator = null;
}
```

To make this work, we need to create the two robots and then tell the BeeperPutter who its collaborator is.

```
RoomMeasurer Kristin = new RoomMeasurer(…);
BeeperPutter Sue = new BeeperPutter(…);
Sue.learnCollaborator(Kristin);
Kristin.measureRoom();
Sue.putBeepers();
```

Notice that the reason for wanting to send an object as a parameter is that the receiver needs to communicate with that object.

It is possible, by the way, to define several methods in the same class with the same name, provided that they have different lists of parameter types. These methods are as different as if you had given them separate names. So, for example, you could write a method

```
void move(int distance)
{
  for(int i = 0; i < distance; ++i)
  {
        move();
  }
}
```

in any class and there will be no confusion.

9.5 Other Classes

In Chapter 8 we saw a use of the Die class. Here is the file in which it is defined. It introduces a few new concepts.

```
package cs1;
import java.util.Random;
import java.lang.Math;
public class Die
{     private int value;
      private Random r = new Random();

      public Die(int faces)
      {     value = faces;
      }

      public int roll()
      {     return Math.abs( r.nextInt()) % value + 1;
      }

      public int faces()
      {     return value;
      }
}
```

Everything we have done up to now has been in a single Java package called karelthe robot. We have had a package statement at the beginning of each of the files we have written. The Die class, which implements a simple kind of randomization, is in a different package called cs1. This package contains a number of classes that can be used to learn important concepts of computer science. The Die class also needs some classes from other packages. In particular, it needs the Random class from the package java.util and the Math class from java.lang. To get access to a class defined in a different package we can import the class as shown here. Likewise, to use the Die class of package cs1 from outside this package, we should say import cs1.Die;

The Die class has two instance variables. The first, value, is the number of faces of the Die. The other is an instance of the Random class from package java.util. A Random object is a general purpose random number generator. It can be used to generate various kinds of numbers randomly. In our roll method we ask it to generate a random int. We then divide this by the value (number of faces) and retain only the remainder of the division with the % operator. The division operator itself is the slash character, /, by the way, and multiplication is the asterisk: *. Since a remainder is zero or more, but less than the divisor, we then add one to get a value in the range 1…value. Also, the nextInt might return a negative value so we apply the abs (absolute value) method from Math.

This last method is special, since Math is a class and not an object. There are certain messages that we send to classes and not objects. These are called static methods. The Math class has something like the following:

```
public class Math
```

```
{       …
        public static int abs(int x){…}
        …
}
```

A static member of a class is shared by all objects in the class, and if public is shared everywhere in the program. Variables can also be static, in which case they are called static variables or shared variables, rather than instance variables. Static variables don't have separate copies in each object of the class, but the one variable is shared by all, so that if one object changes the value, others can see the changes.

The Java system comes with twenty or more packages, hundreds of classes and thousands of methods. It is one of the most extensive libraries associated with any language. The package java.lang has support classes for the language itself, such as String. Java.util has several classes that are widely used in programming projects. There are other packages for graphical user interface programming, network programming, and sophisticated input to and output from the computer.

9.6 Still More Java

So far, our robots remember if they have any beepers, but they don't remember or have a simple way to know how many they have. We can correct this by building a class that remembers how many it was given when it was created and changes this value whenever it picks or puts a beeper. We will modify BrainyRobot to show how to do this.

```
class BrainyRobot extends Robot // new version
{       public BrainyRobot (String name, int street, int avenue, int
direction, int beepers)
        {       super(street, avenue, direction, beepers);
                myName = name;
                myBeepers = beepers;
        }

        public void pickBeeper()
        {       super.pickBeeper()
                beepers = beepers + 1;
        }

        public void putBeeper()
        {       super. putBeeper ()
                beepers = beepers - 1;
        }
        public int beepers() { return myBeepers; }
        public String name() { return myName; }

        private String myName;
        private int myBeepers;
}
```

Now each BrainyRobot has its own name and it also keeps track of the number of beepers in its own beeper bag. We therefore give the class a new method so that we can ask a BrainyRobot how many

beepers it has. We can use this new service in a variety of ways. We have already seen that we can print out information and we could print this out also.

```
System.out.println("The robot has " + kjr.beepers() + " beepers.");
```

If you are a careful reader you noticed that we have added a String and an int. Java will decide that this makes no sense on its face. We can add ints using addition and we can "add" Strings using concatenation, but no more. Java treats this as a special case and will automatically translate the int to a String for us. Therefore this is concatenation.

However, we can do much more interesting things with such methods. Notice that the beepers method returns a value, just as our predicate methods do, though of different type. Suppose our robot kjr is on a corner with lots of beepers. We could say something like:

```
while (kjr.beepers() < 10)
{     kjr.pickBeeper();
}
```

If there are enough beepers on the corner, kjr will end this fragment with at least 10 beepers. This works because the expression kjr.beepers() < 10 has a boolean value, which is all that is required by if and while statements. Of course, The above statement isn't safe, since we don't know how many beepers there are on the corner. We could improve it as follows:

```
while (kjr.beepers() < 10)
{     if(kjr.nextToABeeper())
      {     kjr.pickBeeper();
      }
}
```

There is a shorthand for this however. It is the && (and) operator.

```
while (kjr.beepers() < 10 && kjr.nextToABeeper())
{     kjr.pickBeeper();
}
```

Here we have made two tests in the same while statement and connected them with and. If the first test fails (kjr has 10 or more beepers already) then we don't execute the body. If it succeeds we do the second test. We only execute the body if the second test also succeeds. There is a similar || (or) operator. The and and or operators always work on boolean values. Java also has comparison operators for the simple data (like int) These are <, >, <=, >=, !=, and ==. The last two are "not equal to" and "is equal to" respectively.

9.7 Java Vectors

When one robot, Alexander, needs to send messages to another robot, Watson, it needs a name or reference by which to refer to the other robot. We have seen that such a reference to Watson could be passed as a parameter to Alexander and then Alexander could remember the reference in an instance variable so that messages could be sent. If Alexander needs to send messages to one or two robots then this solution can be repeated, but it is less convenient if there are many. It is actually not possible if we don't know in advance how many robots Alexander must talk to, since then we wouldn't know how many such variables to define.

Java provides a solution to this problem in its java.util package with a class called Vector. A Vector is a container of references to other objects. It maintains these references in linear, sequential, order, much like a shopping list, with a first element, a second element, etc. We can put references to objects into it with its addElement method. These new references are inserted at the end, or last, position. Therefore, if Alexander needs to talk to lots of robots, then we give it a method that allows another robot to send its name and Alexander puts the reference into its Vector instance variable.

In Alexander's class we need something like

```
public void rememberMe(ur_Robot name)
{     myVector.addElement(name);
}
...
Vector myRobots = new Vector();
```

Then, whenever any robot sends the message

```
Alexander.rememberMe(this);
```

the reference to the sending robot (who is "this") is stored in Alexander's vector.

Now, Alexander can communicate with each of the robots in its vector. There are two ways to do this. We show the less preferred way first.

```
for(int i = 0; i < myRobots.size(); ++i)
      ((ur_Robot)myRobots.elementAt(i)).move();
```

Here we use a counting loop controlled by the variable i, which will take on values from zero up to but not including the size of the vector. For any value of i in this legal range, myRobots.elementAt(i) is the i'th element in the vector starting from the first, which has index 0. If myRobots.size() is 5, then myRobots has elements at positions 0, 1, 2, 3 ,4, but not at 5.

However, myRobots has type Vector, which the system knows contains Objects, but that is all the system knows about it. We know that it contains robots, but the system does not. If we have an object x, then x.move() would probably illegal, as most objects don't have a move method as robots do. Therefore we must inform the system that what it will find in the vector is an instance of ur_Robot. This is the reason for the cast

```
(ur_Robot)myRobots.elementAt(i)
```

which assures the system that we have an ur_Robot at that position in the vector. The extra set of parentheses is required since we want to send a message to that robot and the parentheses group all of the pieces together into one that represents a robot

```
((ur_Robot)myRobots.elementAt(i) )
```

Casting is often necessary when we put a reference into some container. When we take it out again, the system ususally only knows that the container contains objects and we need to specify the more specific type of the object. Notice also, that our cast is to ur_Robot. If we knew that we had only put Racer robots into the vector, then we could have cast to that instead. You don't need to cast to the most specific type,

however. Here, any kind of robot is at least an ur_Robot, so the cast can't fail. It will be checked by the Java system in any case.

There are some errors that can be made with the above. If you start with 1 instead of 0 you will miss one item in the vector. If you try to extract an element at location size() or beyond, the system will notice the error when the program runs and it will be halted. If you forget the cast, then the Java translator will complain that Objects don't have a move method. If you forget to increment i, with ++i, the loop can run forever, returning the element at location 0.

Java containers provide a different and preferred way to access the elements of the container. Java uses Enumeration objects to retrieve the objects in a container one at a time. Here is the preferred way to send all of the robots in Alexander's vector the move message.

```
Enumeration e = myRobots.elements();
while(e.hasMoreElements())
      ((ur_Robot)e.nextElement()).move();
```

First notice that we still have to cast. Here, the vector itself gives us an enumeration object when we call its elements method. This object is capable of "enumerating" or listing all of the elements currently in the vector, though it would be an error to try to change the vector while we are doing this. To list the elements and apply messages to them, we first call the enumeration's hasMoreElements method to determine if there are any elements yet to list. If this returns true then we can call its nextElement method to obtain the next element in the vector. We can then cast it to any of its legal types and apply messages as desired.

This technique of getting an enumeration from a container and then using a while loop to obtain the elements is a standard Java idiom that is used in many places.

One last note about vectors in Java. A Vector can hold any number of references to objects. From one or two, to thousands or more. It does so quite efficiently, by growing and shrinking its internal "memory" as we insert (and remove) objects. You can remove the last element with the message

```
myRobots.removeElementAt( myRobots.size() - 1 );
```

9.8 Java Arrays

If we have a situation in which one robot needs to be able to speak to a large, but fixed number of other robots we can use Java arrays to help us. An array has some of the properties of a Vector, but it has fixed size and uses different syntax to manipulate it. It is a kind of container that contains other things. It can contain references to objects, or it can contain more primitive things like ints and booleans.

Consider a situation in which we want to create 20 Racer robots, one each of the first 20 streets, all facing east. We could think of 20 names, of course, Racer_1, Racer_2, ... but this is tedious and ugly. Instead we can use an array to hold the references to the robots, rather than individual reference variables.

```
Racer [] racers = new Racer[20];
```

On the left side of the equals sign we say that we want a variable named racers that will refer to an array of racers, not to an individual racer. Racer is the kind of element that the array shall have. On the right

side we actually create the array and say that it shall have 20 elements (of type Racer). This creates the array, but it doesn't create any robots. We need to call new 20 times for the robots as well.

```
for(int i = 0; i < 20; ++i)
       racers[i] = new Racer(i, 1, East, 0);
```

The individual Racers don't have names. Instead we use the subscript expression racers[3] to refer to the Racer in the slot with index 3. Like Vectors, the slots are numbered from 0, but unlike Vectors, the access uses "subscript" notation, [], rather than an elementAt method. However, the effect is the same. The reason for the difference is that arrays are part of the Java language, while Vectors are part of the Java libraries: Vector is just a class, such as you could write yourself.

Once we have created all of the racers we could send all of them the race message with:

```
for(int i = 0; i < 20; ++i)
  racers[i].race();
```

Arrays don't have enumerations. They are usually manipulated directly using for loops as shown here. You can make errors with arrays as with vectors. The legal indices of the array are from 0 to one less than the size. If you give an index outside those bounds it will be an error that will be caught as the program runs. Notice that we have used the number 20 in three places in the above program fragments. This is error prone, since if the problem changes and we only need 10 Racers and edit some of these but miss others we will have errors. There are two things we should do here. First we should give a name to the number of robots so that we can easily find the thing that needs to change.

public static final int numberOfRobots = 20;

A final value in Java can't be changed. A static value is shared among all elements of the class. A static final int is just a constant int that can be seen by all members of its class, and if it is public also, can be seen everywhere. We could then replace all of the 20's in the above with this new name. For example:

```
for(int i = 0; i < numberOfRobots; ++i)
        racers[i] = new Racer(i, 1, East, 0);
```

There is another solution to this, that works for all but the creation of the array itself. Once an array is created it knows its own length, and you can ask an array for its length:

```
for(int i = 0; i < racers.length; ++i)
        racers[i] = new Racer(i, 1, East, 0);
```

Length is an instance variable of every array. You can look at its value as here, but you can't change it.

Here is another, somewhat frivolous, use of arrays that shows another important option. Suppose we want to build a robot that "speaks" fortune cookies. Each time we ask it to speak it prints out a fortune cookie on System.out. We would want it to know a fairly large number of different "fortunes" and to give them out randomly. We can use a Die object for the random part, and an array of Strings to hold the fortunes. Here we show a static array, so that all such robots have the same fortunes. We also define the contents of the array by putting the values in the respective cells (Strings) between braces. We don't call new to create

this array. The array on the right hand side exists already by reason of our writng out this compex expression. However, like any array cookies knows its own length.

```
	private static String [] cookies =
	{	"The skin is toughter than the bananna.",
		"Ketchup travels at 25 miles per year.",
		"Have a nice day, anyway.",
		"May all your children be above average."

	};

	private Die die = new Die(cookies.length);
```

With these definitions, the speak method is quite easy to write:

```
public void speak()
{	System.out.println( cookies[ die.roll() - 1 ] );
}
```

Note that the die produces a roll between 1 and its number of faces which is the length of the array. We want a subscript between 0 and one less than the length, so we subtract 1 from the roll. Notice that we let the array count itself, since we used its length rather than a value like 4, which might change if we think of some new cookies and add them.

9.9 Important Ideas From This Chapter

string
vector
array
static method
final class
final method

9.10 Problem Set

1. Write and test a new class, OdometerRobot, which keeps track of how far it has moved since it was created and can tell us how far that is.

2. Examine the last three code fragments in Section 9.6. What happens in each of them if kjr starts on a corner with no beepers?

3. Create a Strategy class (see Chapter 4) that can be used to place any given number of beepers on a corner. How many beepers is determined when the strategy is created and is never changed afterwards.

4. Build an observable robot (see Chapter 4) that can handle any number of observers. The register method should add a new observer to a Vector that it remembers. When the key method is executed, all observers are sent an action message. Use an Enumeration over the Vector to do this. Test your class with at least three different observers.

5. Build a Spy class that works as follows. In the first phase the robot collects two clues from each accomplice. The first clue it just saves in a vector, and the other it uses to find the next accomplice. At the end, the Spy returns to the origin and only then follows all of the saved clues, in the order it got them, to find the treasure.

6. Re read Problem 12 of Chapter 4. Suppose we want the robot to apply its initial strategy exactly three times and then permanently change to its second strategy. Implement this by changing the strategy objects. If you have a good solution to Problem 12 from Chapter 4, you should only need to write a single new Strategy class.