Karel J Robot Chapter 4

Robots have the ability to survey their local environment and then decide from that information what to do next. They can be programmed using if, if-else, if-elseif-elseif-else, &&, || statements.

There is a class **Robot** that inherits from the **UrRobot** class. We will now make our SuperRobot class extend from that Robot class. The **Robot** class is much improved because it has the following additional methods:

```
public boolean frontIsClear()
{...}
public boolean nextToABeeper()
{...}
public boolean nextToARobot()
{...}
public boolean facingNorth()
{...}
public boolean facingSouth()
{...}
public boolean facingEast()
{...}
public boolean facingWest()
{...}
public boolean anyBeepersInBeeperBag()
{...}
```

All these methods return either true or false since they are boolean methods. nextToABeeper method returns true when a robot is on the same corner as one or more beepers. nextToARobot method returns true when there is another robot on the same corner. To return the negative form, use the not operator (!).

```
import kareltherobot.*;

public class Test implements Directions
{
  public static void main(String [] args)
  {
      World.readWorld("Test.kwld");
      World.setVisible(true);
      World.setDelay(10);
      SuperRobot karel = new SuperRobot(2,5,North,4);
      if (karel.nextToABeeper())
      {
          karel.pickBeeper();
      }
      if (!karel.frontIsClear())
      {
          karel.turnAround();
      }
  }
}
```

The form of the if/else is similar to the *if* instruction, except that it includes an *else* clause. The robot first determines whether the Boolean expression is true or false. If the expression is true, the robot executes the instructions inside the { } after the *if* statement. If the expression is false, the robot executes the instructions inside the { } after the *else* statement. Note the code below is just a section of code.

```
SuperRobot karel = new SuperRobot(2,5,North,4);
if (karel.frontIsClear())
{
    karel.move();
}
else
{
    karel.jumpHurdle();
}
```

if/else is the syntax programmers use when there are two courses of action. We can extend that to *if/elseif/else* when there are three courses of action. We can extend that further to *if/elseif/elseif/else* when there are four courses of action. You can keep adding more elseif statements for as many courses of action as you'd like.

```
SuperRobot karel = new SuperRobot(2,5,North,4);
if (karel.frontIsClear())
{
    karel.move();
}
else if (karel.nextToABeeper())
{
    karel.pickBeeper();
}
else
{
    karel.turnLeft();
}
```

The *else* statement covers all situations that were not identified earlier in the *if/elseif* statements. There does not have to be an *else* statement.

```
SuperRobot karel = new SuperRobot(2,5,North,4);
if (karel.frontIsClear())
{
    karel.move();
}
else if (karel.nextToABeeper())
{
    karel.pickBeeper();
}
```

Note which lines end in semicolons and which do not. What is the rule for when you type semicolons at the end of a line?

AND Operator
You can join multiple Boolean expressions together using the AND operator which is two ampersands (&&). For the resulting expression to be true, all the parts need to be true.

```
if (!karel.frontIsClear() && !karel.rightIsClear())
{
    karel.turnLeft();
}
```

Here is the truth table for the AND operator:

!karel.frontIsClear()	!karel.rightIsClear()	AND
T	T	T
T	F	F
F	T	F
F	F	F

OR Operator
You can join multiple Boolean expressions together using the OR operator which is two pipes (||). For the resulting expression to be true, one of the parts need to be true.

```
if (karel.facingNorth() || karel.facingSouth())
{
    karel.faceNorth();
}
else
{
    karel.faceEast();
}
```

Here is the truth table for the OR operator:

karel.facingNorth()	karel.facingSouth()	OR
T	T	T
T	F	T
F	T	T
F	F	F

Problem Set

1. In your SuperRobot class make four methods called faceNorth(), faceSouth(), faceEast(), faceWest(). These methods will face the robot in the specified direction no matter which direction he is initially facing. There already is a class TestFace that starts 4 robots off that are facing in different directions and then turns them all in the specified direction. Change the comment markers in that code to test the various directions.

   The robot class only comes with a frontIsClear() method. Write 3 more methods in your SuperRobot class called rightIsClear(), leftIsClear(), and backIsClear(). These methods check for walls in those new directions. Try them by running the TestClear class and read the Terminal Window to check the accuracy of the prints. Make sure the robot is back in its original direction after performing each of the methods. Here are samples of how printing is done in the Terminal Window.

   ```
   System.out.println("Clear on right");
   System.out.println("Number of beepers is " + variable);
   ```

2. SurroundedByWalls: Write a class that turns a robot off if the robot is completely surrounded by walls, unable to move in any direction. There exists a turnOff method in the UrRobot class that you are to use only in this problem. You may not use the turnOff method in any other labs. If the robot is not surrounded, leave itself turned on and remain on the same corner, facing the same direction in which it started. Make 5 robots all with different numbers and configurations of surrounding walls. No new SuperRobot methods.

3. SteeplechaseRace: The robot should be able to climb a wall that is one, two, or three blocks high. Make a new class SteeplechaseRace in which the robot runs an eight block long steeplechase. This class should have the robot make the choice to move or climb at each of the 8 corners as the robot travels on 1st Street. Make a climbWall() method in the SuperRobot class. (Hint: In the climbWall method, get the robot to the top of any height wall before continuing over and down the wall.)

this method there must be no beepers left on the corner. Name this method findNextDirection.

4. Write another version of findNextDirection(see the previous problem). In this version the robot mus eventually face the same directions, but it also must leave the same number of beepers on the corner as were there originally.

5. Write an instruction that turns a robot off if the robot is completely surrounded by walls, unable to move in any direction. If the robot is not surrounded, it should execute this instruction by leaving itself turnec on, and by remaining on the same corner, facing the same direction in which it started. Name this instruction turnOffIfSurrounded. Hint: To write this instruction correctly, you will need to include a turnOff inside it. This combination is perfectly legal, but it is the first time that you will use a turnOff instruction outside the main task block.

6. Program a robot to run a mile long steeplechase. The steeplechase course is similar to the hurdle race, but here the barriers can be one, two, or three blocks high. Figure 5-5 shows one sample initial situation, where the robot's final situation and path are shown on the right. Call the class of this new robot Steeplechaser. It should have Racer as a parent class. Override appropriate instructions of Racer to implement the new behavior.

Initial Situation — Final Situation and Robot's Path

FIGURE 5-5 Steeplechase Race Task

7. Rewrite and check the following new instruction, taking care to interpret all of the robot programming grammar rules correctly. This instruction uses nested IF'S to face a robot toward the east; verify that it is correct by simulation. Hint: When trying to check this instruction, ignore the instruction indentation. One way of doing this is to have someone read you the instruction. While they are reading the instruction, you should keep track of the meaningful components of the instruction. This is exactly what the factory does when it reads instructions to robots at the factory.

```
public void faceEast()
{
        if(! facingEast())
{
                if(facingWest()){
                        turnLeft();
                        turnLeft();}
```

4. MazeWalker: Write an additional method in your SuperRobot class called followWallRight(). This method assumes that whenever the robot executes this instruction there is a wall directly to its right. This method needs to teach a robot how to follow the wall on its right side, no matter what the wall situation. Make a class called MazeWalker that tests the four different possible situations by calling the followWallRight() method once for each of the 4 robots in the 4 situations shown.

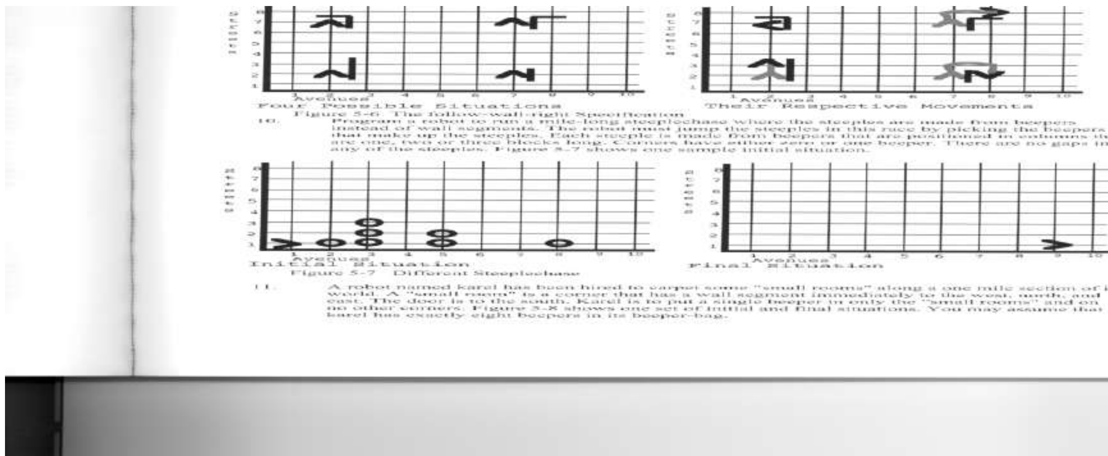

Figure 5-6 The follow-wall-right Specification

10. Program a robot to run a mile-long steeplechase where the steeples are made from beepers instead of wall segments. The robot must jump the steeples in this race by picking the beepers that make up the steeples. Each steeple is made from beepers that are positioned in columns th are one, two or three blocks long. Corners have either zero or one beeper. There are no gaps in any of the steeples. Figure 5-7 shows one sample initial situation.

Figure 5-7 Different Steeplechase

11. A robot named karel has been hired to carpet some "small rooms" along a one mile section of it world. A "small room" is a corner that has a wall segment immediately to the west, north, and east. The door is to the south. Karel is to put a single beeper in only the "small rooms" and on no other corners. Figure 5-8 shows one set of initial and final situations. You may assume that karel has exactly eight beepers in its beeper-bag.

5. CarpetRooms: Karel has been hired to carpet some rooms. A room is a corner that has a wall segment immediately to the west, north, and east. The door is to the south. Karel is to put a single beeper in room and on no other corners. The robot starts with 8 beepers in its beeper-bag. Do not make any SuperRobot methods.

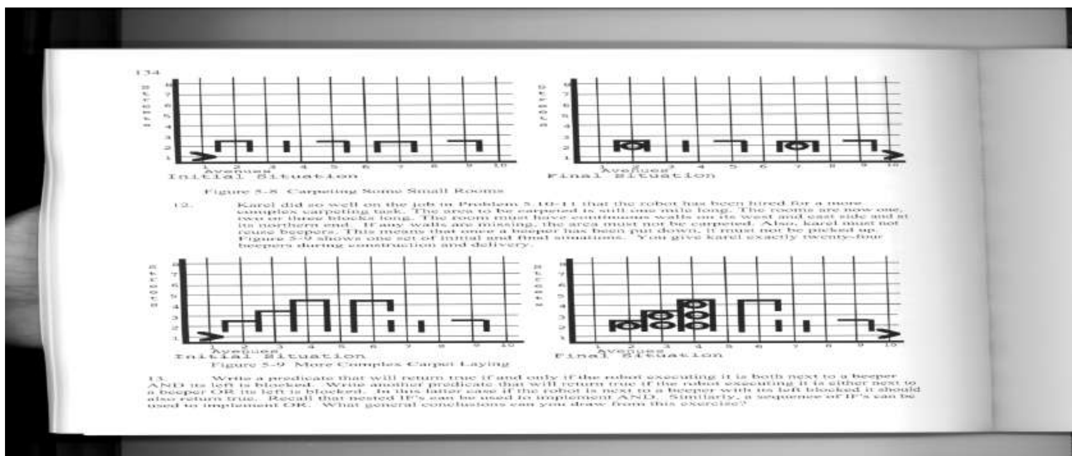
134

Figure 5-8 Carpeting Some Small Rooms

12. Karel did so well on the job in Problem 5.10-11 that the robot has been hired for a more complex carpeting task. The area to be carpeted is still one mile long. The rooms are now one, two or three blocks long. The room must have continuous walls on its west and east side and at its northern end. If any walls are missing, the area must not be carpeted. Also, karel must not reuse beepers. This means that once a beeper has been put down, it must not be picked up. Figure 5-9 shows one set of initial and final situations. You give karel exactly twenty-four beepers during construction and delivery.

Figure 5-9 More Complex Carpet Laying

13. Write a predicate that will return true if and only if the robot executing it is both next to a beeper AND its left is blocked. Write another predicate that will return true if the robot executing it is either next to a beeper OR its left is blocked. In this latter case if the robot is next to a beeper with its left blocked it should also return true. Recall that nested IF's can be used to implement AND. Similarly, a sequence of IF's can be used to implement OR. What general conclusions can you draw from this exercise?