Discussion
Loops and If

Any for-loop can be re-written as a while-loop.

```
for(int k = 1; k <= n; k++)
   karel.turnLeft();
```

```
int k = 1;
while(k <= n)
{
   karel.turnLeft();
   k++;
}
```

Both loops as written will try to make karel turn left for *n* times, no matter what. However, it has become conventional to use the for-loop for definite loops and the while-loop for indefinite loops, where you don't know beforehand exactly how many times to loop.

The while-loop below is a better example of an indefinite loop, for we don't know when it will stop. It keeps picking up a beeper as long as `nextToABeeper` is `true`. The result is to make a robot that is standing on top of a pile to pick up that entire pile of beepers.

```
while(karel.nextToABeeper())
   karel.pickBeeper();
```

An if-statement is similar to a while-loop, but only makes one decision. The if-statement below will cause karel to pick up one beeper if he is currently standing on top of one. If karel is not currently standing on top of a beeper, the code just moves on. Thus, the if-statement acts as a guard to prevent the program from crashing in certain situations. In other words, an if-statement checks its condition only once.

```
if(karel.nextToABeeper())
   karel.pickBeeper();
```

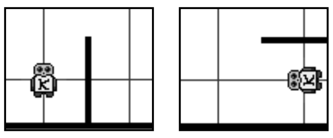
Warning

There is no such thing as an if-loop. Don't say "if-loop," say "if-statement."

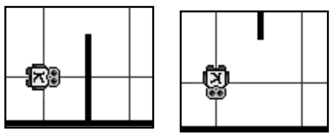
Java distinguishes between *void methods* and *return methods*. Void methods take action and change the situation, such as `public void turnRight()`. In contrast, return methods provide information about a robot's situation. Since there are different kinds of information, return methods come in several varieties. One variety returns *boolean* values, either true or false. Think of booleans as answering yes-no questions. An example of a boolean method, used above, is `public boolean nextToABeeper()`.

Another boolean method defined in the Robot class is `frontIsClear`. It determines whether or not a wall blocks a robot's path. Look at the examples below; you can see that this method makes no distinction whatsoever as to the cardinal-direction in which the robot is facing.

`frontIsClear()` returns `true`



`frontIsClear()` returns `false`



There is no method `frontIsBlocked`. However, there is the exclamation mark (!), called the *not* operator, and follows the rules not-true is false and not-false is true. The `not` operator goes in front of the whole expression, for example, `!karel.frontIsClear()`. We might read it as "not karel dot frontIsClear."

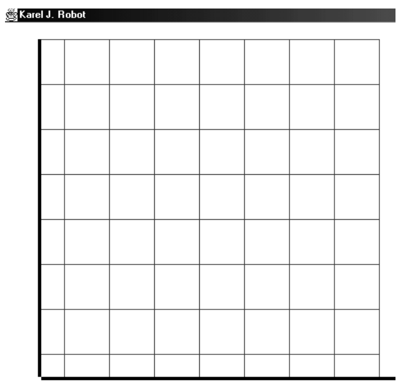
Exercises
Booleans

Look at the Robot's API to see what boolean methods might be helpful here.

1. Write the commands to force a Robot object named `karel` to face west no matter what direction it is initially facing.	2. Write the commands to make an Athlete named `ann` to put down all her beepers.
3. Write the commands for Climber `c` to pick up a pile of beepers.	4. Write the commands for Racer `ray` to stop moving when it is next to (on top of) another robot.

5) Given the declarations and the default map, indicate the boolean value of each statement.

```
Robot pete = new Robot(5, 1, Display.WEST, 37);
Robot lisa = new Robot(2, 6, Display.SOUTH, 0);
```



`!pete.frontIsClear()`	true	false
`pete.hasBeepers()`	true	false
`!lisa.hasBeepers()`	true	false
`!pete.facingWest()`	true	false
`lisa.facingWest()`	true	false

6) The logical AND (in Java) is the symbols **&&**. The logical OR (in Java) is the symbols **||**. Evaluate:

true && true →	true || true →
true && false →	true || false →
false && true →	false || true →
false && false →	false || false →

7) Extra Credit: make a truth table that shows that De Morgan's Laws are true:

```
!(a && b) == !a || !b          !(a || b) == !a && !b
```

Lab06
A Half-Dozen Tasks

Objective

while-loops and if-statements

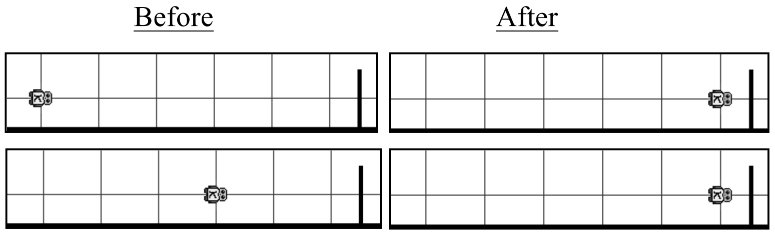
Background

This lab instantiates 6 Robots to perform 6 different tasks. For convenience, we code each task as a class method. Each task uses one or more indefinite loops. Some tasks also use if-statements.

Row	Task
1	Go to the end of the row of beepers.
2	Go to the beeper.
3	Go to the wall.
4	Go to the wall, picking up all the beepers (max one beeper per pile).
5	Go to the wall, picking up all the beepers.
6	Go to the end of the row of beepers; there is one gap somewhere.

If `temp` (for temporary) is a Robot object, the solution to task number three is:

```
while(temp.frontIsClear())
{
  temp.move();
}
```



This loop will repeat an unknown number of times, as long as temp's path is not blocked. The pictures show two such situations. Warning: if there is not a wall somewhere along the path then this segment of code will repeat forever; this is called an *infinite loop* and your program will never end.

This is the first lab in which your program must work correctly for many different robot-maps, all of which have the same basic structure. Thus, the name of the map will not be *hard-coded*:

```
Display.openWorld("maps/tasks1.map");
```

because your program cannot be fully tested with only the `tasks1` map. Instead you must import the javax.swing.JOptionPane class and use the commands:

```
String filename = JOptionPane.showInputDialog("What robot map?");
Display.openWorld("maps/" + filename + ".map");
```

When your program runs, an input dialog box will open prompting you for the name of the map to use. Run your program three times, entering either "tasks1", "tasks2", or "tasks3". Your program does not work unless it runs successfully for all three maps.

Specification

This is your first program that has shell code provided for you. Be sure to load Unit1\Lab06.java. Set the size to 10x10. Notice the 6 class methods, each instantiating a Robot object. Accomplish all six tasks. The same code must work with all three maps "tasks1", "tasks2", and "tasks3".

Extensions

1. Modify task_04 and task_05 so that they count and print the number of beepers. Look at the code on the next page for a hint. In tasks1.map, task_04 will count 4 beepers and task_05 will count 11.
2. Create a valid robot-map for this lab using `MapBuilder.jar` and test your program with that map.

Exercises

Lab06

How can the robot keep count of its beepers? One way is to use an integer variable that starts at zero and increments every time. This code counts and prints the number of beepers in a pile.

```
int count = 0;
while(karel.nextToABeeper())
{
   karel.pickBeeper();
   count++;
}
System.out.println("Count is "+ count);
```

Complete the methods below.

public static void task_07()   //go to the beeper or the wall. Count and report the number of steps you took.
{

}

public static void task_08() //go until you are next to another robot, then put all your beepers down.
{

}

public static void task_09() //put down 5 different piles with 4 beepers in each pile. Use definite loops.
{

}

public static void task_10() //fill in gaps with a beeper. Stop when you reach a wall.
{

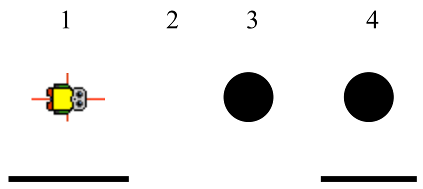
}

public static void task_11() //while there is a wall to your right, put down one beeper at each step
{

}

public static void task_12() //go until there is a wall to your right and you are standing on a beeper
{

}



Using `MapBuilder.jar`, make your own maps for tasks 7 through 12 above.