Control Statements for Karel the Robot (Karel2)

Objective: To learn how to program Karel the Robot so it can repeat tasks and make decisions.

Background:
In the previous exercise, we learned about Karel the Robot and how to make Karel perform some rudimentary tasks. Although we learned how to decompose and define new methods for Karel, it cannot solve any new problems on its own. Previous tasks took less than 30 commands. Suppose a problem takes Karel over 1000 commands to complete. There is a point where decomposition will not reduce the code very much, so you end up with a very long program that is difficult to read. Writing step-by-step linear programming only helps us up to a point.

To solve more difficult problems, it would help if Karel could make some decisions on his own. It would also help if repetitive tasks (e.g. `move(); move(); move();` etc) could be expressed in a more succinct way. Fortunately, Karel's world provides us statements that help to affect the order in which the program executes. These are called **control statements** and they generally fall into two categories:

1. ***Conditional statements***. Conditional statements specify that certain statements in a program should be executed only if a particular condition holds. In Karel's world, you specify conditional execution using an **`if`** statement.
2. ***Iterative statements***. Iterative statements specify that certain statements in a program should be executed repeatedly, forming what programmers call a **loop**. Karel's world supports two different iterative statements: a **`for`** statement that is useful when you want to repeat a set of commands a predetermined number of times and a **`while`** statement that is useful when you want to repeat an operation as long as some condition holds.

Conditional statements:
We will be working in the directory **Karel1and2** created in the previous exercise. Let's revisit the **`fillPothole`** method from last time.

Before filling in the pothole, Karel might want to check if some other repair crew has already filled in the hole with a beeper. If so, Karel does not need to put down a second beeper. To represent such a check in the context of a program, we would use the **`if`** statement. The **`if`** statement takes on the following form.

```
if ( conditional test ) {
    statements to be executed only if the condition is true
}
```

The conditional test shown in the first line must be replaced by one of the tests Karel can perform on his environment. The result of the test is either true or false. If the test is true, Karel executes the statements enclosed in braces; if the test is false, Karel does nothing.

The 18 conditional tests in Karel's world are listed in the table below.

Test	*Opposite*	*What it checks*
`frontIsClear()`	`frontIsBlocked()`	Is there a wall in front of Karel?
`leftIsClear()`	`leftIsBlocked()`	Is there a wall to Karel's left?
`rightIsClear()`	`rightIsBlocked()`	Is there a wall to Karel's right?
`beepersPresent()`	`noBeepersPresent()`	Are there beepers on this corner?
`beepersInBag()`	`noBeepersInBag()`	Any there beepers in Karel's bag?
`facingNorth()`	`notFacingNorth()`	Is Karel facing north?
`facingEast()`	`notFacingEast()`	Is Karel facing east?
`facingSouth()`	`notFacingSouth()`	Is Karel facing south?
`facingWest()`	`notFacingWest()`	Is Karel facing west?

Notice that each test has a corresponding opposite test. For example, you can use **`frontIsClear()`** condition to determine if the path in front of Karel is clear, or the **`frontIsBlocked()`** condition to determine if the path in front is blocked by a wall. Both conditions are provided and you can decide which is easiest to apply.

We can use the **`if`** statement in the **`fillPothole`** method so Karel only puts down a beeper if there is not already a beeper in the hole. A new definition of **`fillPothole`** would look like this:

```
public void fillPothole() {
   turnRight();
   move();
   if (noBeepersPresent()) {
      putBeeper();
   }
   turnAround();
   move();
   turnRight();
}
```

By convention, the body of any control statement is indented with respect to the braces that surround it. Indentation makes it much easier to see exactly which lines are being addressed by the control statement. You may want to put in more checks, like to check if Karel has any beepers in his bag before he puts down a beeper.

```
if (noBeepersPresent()) {
   if (beepersInBag()) {
      putBeeper();
   }
}
```

When control statements are inside other control statements, the statements are said to be **nested**. In this statement the **`putBeeper`** command is executed only if there is no beeper on the corner and there is at least one beeper in Karel's bag.

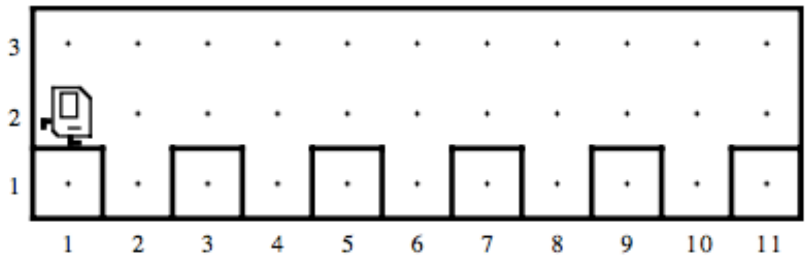
Sometimes the outcome of a decision is either do something when a condition is true, or do something else if it is false. For these cases, the **`if`** statement in Karel's world has an alternate form that looks like this:

```
if ( conditional test ) {
   statements to be executed only if the condition is true
} else {
   statements to be executed only if the condition is false
}
```

We will have an example of this type of **`if`** statement later in our discussion.

Iterative statements
In solving Karel problems, you will often find that repetition is a necessary part of your solution. If you were really going to program a robot to fill potholes, it would hardly be worthwhile to have it fill just one. The value of having a robot perform such a task comes from the fact that the robot could repeatedly execute its program to fill one pothole after another.

To see how repetition can be used in the context of a programming problem, consider the following roadway in which potholes are spaced along 2nd Street at every even numbered Avenue.



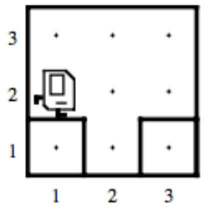
Our current **`PotholeFillingKarel`** will fill in one pothole, but this problem has five equally spaced potholes. Luckily for us, Karel's world supports the **`for`** statement, which specifies exactly how many times you want Karel to perform the same function.

```
for (int a = 0; a < count ; a++) {
    statements to be repeated
}
```

In the **`for`** statement, count is an integer indicating the number of repetitions. Let's make a new file called **RoadRepairKarel.java** and copy in the contents of the **`PotholeFillingKarel`** class. Be sure to change the name of this new class to **`RoadRepairKarel`** to match the filename. To implement repetition, all we need to do is change the **`run`** method as follows:

```
public void run() {
   for (int a = 0; a < 5; a++) {
      move();
      fillPothole();
      move();
   }
}
```

The **`for`** statement is only useful when you know in advance exactly how many times a task is to be repeated. What if we had a different world with a different number of potholes? For example, this world:



Run **`RoadRepairKarel`** and click the "**Load World**" button. In the file window, select **RoadRepairKarel3x3.w**. You should get the world shown above. Next click "**Start Program**" and you will get an error! `RoadRepairKarel` only knows how to work on a road with exactly 5 potholes, so it is not very flexible.

To be more flexible, Karel's world provides us with a new iterative statement that can handle different situations, the `while` statement.

```
while ( conditional test ) {
    statements to be repeated
}
```

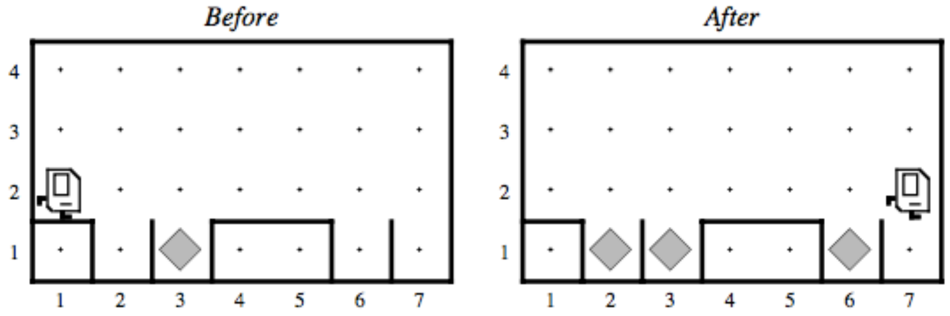
Karel now needs to check if his path is clear in front of him. The condition `frontIsClear()` in a `while` statement will insure that Karel repeatedly fills potholes until he detects a wall. This methodology allows us to handle a number of different roadways including **RoadRepairKarel3x3.w** above. Exchange the `for` statement with the `while` statement in `RoadRepairKarel` as follows:

```
public void run() {
   while (frontIsClear()) {
      move();
      fillPothole();
      move();
   }
}
```

Whether we load the world **RoadRepairKarel.w** or **RoadRepairKarel3x3.w**, Karel will fill in all the evenly spaced potholes he finds.

Assignment:

1. Copy your `RoadRepairKarel` program into a new file **UnevenRoadRepairKarel.java** and change the class name to `UnevenRoadRepairKarel`. Solve the following problem.



Do not fill holes that are already filled! That would put 2 beepers in a hole, a waste of resources. Remember, only one beeper per pothole.

2. Modify `UnevenRoadRepairKarel` to fix that last pothole at the corner of 1st Street and 7th Avenue. Leaving the last pothole unfilled is called a **fencepost error**. Often solutions handle a repetitive problem, like potholes, except the first or the last occurrence of the problem. The fix is often simple, but finding a fencepost error is not always easy to detect. In fact, you will find this error to be very common in programming and is often very costly in time to discover and fix.