# 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. `moveForward(); moveForward(); moveForward();` 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 **`repeat`** 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**:
Let's revisit the **`fillTheHoles`** (1.2.2) method from last time.

Before filling in the hole, 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 5 conditional tests in Karel's world are listed in the table below.

| Shortcut | Condition | Meaning |
|---|---|---|
| F7 | `onBeeper()` | Karel checks whether a beeper is on the square he currently stands on. |
| F8 | `beeperAhead()` | Karel checks whether a beeper is on the square immediately in front of him. |
| F9 | `leftIsClear()` | Karel checks whether no wall is between him and the square to his left. |
| F10 | `frontIsClear()` | Karel checks whether no wall is between him and the square in front of him. |
| F11 | `rightIsClear()` | Karel checks whether no wall is between him and the square to his right. |

In addition to these conditional tests, there are three logical operators available.

| | |
|---|---|
| **!** | not |
| **&&** | and |
| **\|\|** | inclusive-or |

You may use these logical operators with the conditions. The following table describes how they work.

| Condition | Meaning |
|---|---|
| `!a` | holds if *a* does **not** hold (and vice versa) |
| `a && b` | holds if both *a* **and** *b* hold |
| `a \|\| b` | holds if *a* **or** *b* (or both) hold |
| `a \|\| !b && c` | `a \|\| ((!b) && c)` |

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

```
void fillHole() {
   turnRight();
   moveForward();
   if (! onBeeper()) {
      dropBeeper();
   }
   turnAround();
   moveForward();
   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. When control statements are inside other control statements, the statements are said to be **nested**. Luckily for us Karel's programming environment automatically indents to make our program readable.

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:

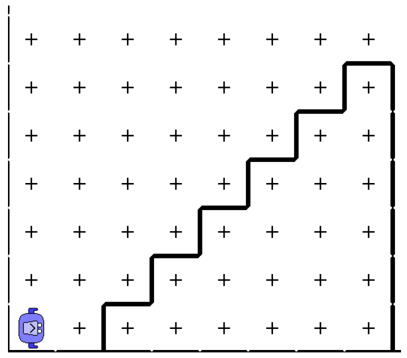
```
if ( conditional test ) {
   statements to be executed only if the condition is true
} else {
   statements to be executed only if the condition is false
}
```

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

## 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 holes, 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 problem **climbTheStairs** (1.2.1) shown below. Climbing each step will take the same series of moves. To get to the top will require repeating the same commands over and over. Even decomposition will require calling the same method over and over.



Luckily for us, Karel's world supports the **`repeat`** statement, which specifies exactly how many times you want Karel to perform the same function.

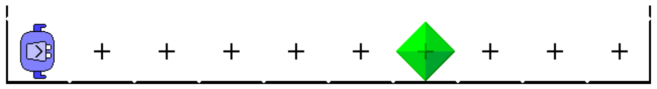
```
repeat ( count ) {
   statements to be repeated
}
```

In the **`repeat`** statement, *count* is an integer indicating the number of repetitions. Let's try to solve the **`climbTheStairs`** problem. To implement repetition, all we need to do is repeat the same function six times for six steps.

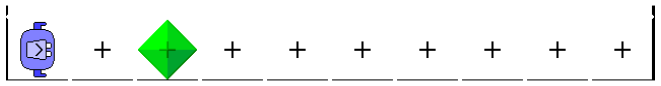
```
repeat (6) {
   climbStep();
}
```

I will leave it to you to implement the **`climbStep()`** method.

The **repeat** statement is only useful when you know in advance exactly how many times a task is to be repeated. What if we wanted to have Karel find a beeper but we did not know beforehand how far Karel needs to go. For example, there could be a world like this:



Or a world like this:



Some Karel problems test your code on thousands of situations and you must pass them all. In this case we cannot program a specific number of steps, so **repeat** will not suffice. We need to write a program that will adapt to the circumstances.

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 can now check to see if he is on the beeper. The condition **onBeeper()** in a **while** statement will insure that Karel repeatedly moves forward until he is on top of the beeper. This methodology allows us to handle different distances. The **while** statement will appear in our program as follows:

```
while (! onBeeper()) {
   moveForward();
}
```

This solves our many-worlds problem. If we know exactly how many iterations then we use **repeat**. If we iterate based on a condition then we use **while**.

**Assignment**:
You will solve the following four problems. The Karel environment stores your program in a file **karel.txt** in your home directory. BEWARE: If you start working on a new problem, the old code is lost.

To be sure that your work is saved properly, it is important that when you finish a problem you should move the **karel.txt** file to a directory you will create called **Solutions**. Once moved, rename the file so you know what it contains. For example, the file containing the solution to **climbTheStairs** should be renamed **climbTheStairs.txt**. Do this for each problem you program.

1. Solve the problems **defuseOneBomb** (1.1.2) and **defuseTwoBombs** (1.1.3). Use the **repeat** command on the first and the **while** command for the second. Decompose repetitive segments of code into their own methods.

2. Solve the problem **`saveTheFlower`** (1.2.3). Decide which iterative command, **repeat** or **while**, works best.

3. Solve the problem **`mowTheLawn`** (1.2.4). This is a slight variation on prior problems with its own twist.

4. Solve the problem **`tileTheFloor`** (1.3.4). This requires a compound conditional that includes a logical operator.