Eric Roberts
CS 106A

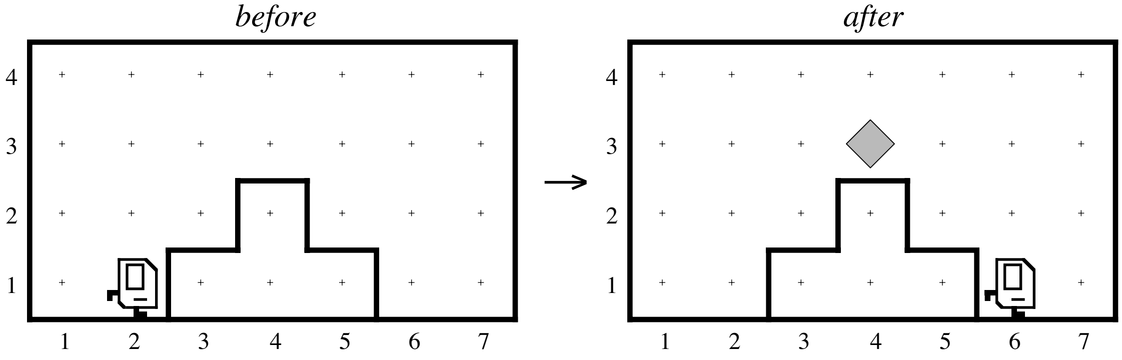
Handout #6
January 6, 2010

Programming in Karel

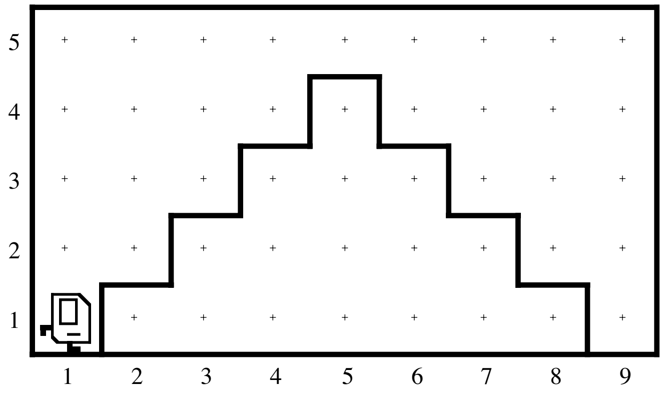
The *Karel the Robot Learns Java* book includes several examples that illustrate the use of control statements in Karel's world. Whenever possible, I like to solve different problems in lecture so that you can see a different set of examples. Today, for example, I will introduce the concept of control statements in the context of teaching Karel to climb a mountain. In the text, you can read a similar introduction in the context of having Karel repair potholes in a roadway. By offering both examples, you are in a better position to understand the general principles as opposed to the details of a particular problem.

In lecture, however, I also want to get you to think about the problems and solve them as we go. If I include the solution in the handout, it's far too tempting just to look at the answer instead of trying to work things out on the fly. Thus, I use these handouts to describe the problems, and then put the solutions up on the web so that you can look over the solutions after class.

The problem we will solve today is that of getting Karel to climb mountains. Like everything else in Karel's world, the mountain is abstract and must be constructed from the available materials, specifically beepers and walls. The goal is to get Karel to climb a mountain marked out by walls, put down a beeper to serve as a flag, and then to climb back down the other side. This problem is illustrated in the following diagram:


before *after*

At first, the goal is simply to solve the specific problem posed by this mountain. From there, however, the more interesting task is to generalize the problem so that Karel can climb larger mountains with the same stair-step structure, like this:



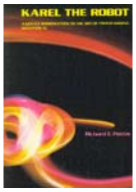
Programming in Karel

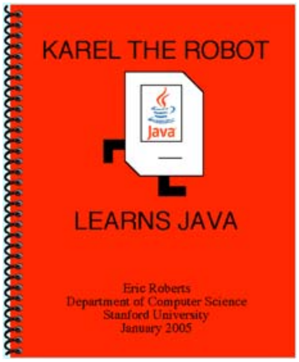
Eric Roberts
CS 106A
January 6, 2010

Once upon a time . . .

Rich Pattis and Karel the Robot

- Karel the Robot was developed by Rich Pattis in the 1970s when he was a graduate student at Stanford.
- In 1981, Pattis published *Karel the Robot: A Gentle Introduction to the Art of Programming*, which became a best-selling introductory text.
- Pattis chose the name *Karel* in honor of the Czech playwright Karel Čapek, who introduced the word *robot* in his 1921 play *R.U.R.*
- In 2006, Pattis received the annual award for Outstanding Contributions to Computer Science Education given by the ACM professional society.





KAREL THE ROBOT

LEARNS JAVA

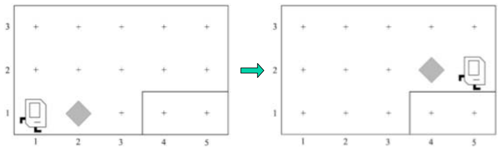
Eric Roberts
Department of Computer Science
Stanford University
January 2005

Review: Primitive Karel Commands

- On Monday, you learned that Karel understands the following commands:

`move()`	Move forward one square
`turnLeft()`	Turn 90 degrees to the left
`pickBeeper()`	Pick up a beeper from the current square
`putBeeper()`	Put down a beeper on the current square

- At the end of class, we designed (but did not implement) a program to solve the following problem:



Our First Karel Program

```
/*
 * File: FirstKarelProgram.java
 * -----------------------------
 * This program moves a beeper up to a ledge.
 */

import stanford.karel.*;

public class FirstKarelProgram extends Karel {

   public void run() {
      move();
      pickBeeper();
      move();
      turnLeft();
      move();
      turnLeft();
      turnLeft();
      turnLeft();
      move();
      putBeeper();
      move();
   }
}
```

Syntactic Rules and Patterns

- The definition of **`FirstKarelProgram`** on the preceding slide includes various symbols (curly braces, parentheses, and semicolons) and special keywords (such as **`class`**, **`extends`**, and **`void`**) whose meaning may not be immediately clear. These symbols and keywords are required by the rules of the Karel programming language, which has a particular ***syntax*** just as human languages do.
- When you are learning a programming language, it is usually wise to ignore the details of the language syntax and instead focus on learning a few general ***patterns***. Karel programs, for example, fit a common pattern in that they all import the **`stanford.karel`** library and define a method named **`run`**. The statements that are part of the **`run`** method change to fit the application, but the rest of the pattern remains the same.

Defining New Methods

- A Karel program consists of ***methods***, which are sequences of statements that have been collected together and given a name. Every program includes a method called **`run`**, but most define ***helper methods*** to you can use as part of the program.
- The pattern for defining a helper method looks like this:

```
private void name() {
    statements that implement the desired operation
}
```

- In patterns of this sort, the boldfaced words are fixed parts of the pattern; the italicized parts represent the parts you can change. Thus, every helper method will include the keywords **`private`** and **`void`** along with the parentheses and braces shown. You get to choose the name and the sequence of statements performs the desired operation.

The `turnRight` Method

- As a simple example, the following method definition allows Karel to turn right by executing three **`turnLeft`** operations:

```
private void turnRight() {
   turnLeft();
   turnLeft();
   turnLeft();
}
```

- Once you have made this definition, you can use **`turnRight`** in your programs in exactly the same way you use **`turnLeft`**.
- In a sense, defining a new method is analogous to teaching Karel a new word. The name of the method becomes part of Karel's vocabulary and extends the set of operations the robot can perform.

Helper Methods in a Program

```
import stanford.karel.*;

public class ImprovedFirstKarelProgram extends Karel {
   public void run() {
      move();
      pickBeeper();
      move();
      turnLeft();
      move();
      turnRight();
      move();
      putBeeper();
      move();
   }

   private void turnRight() {
      turnLeft();
      turnLeft();
      turnLeft();
   }
}
```

Exercise: Defining Methods

- Define a method called **`turnAround`** that turns Karel around 180 degrees without moving.
- Define a method **`backup`** that moves Karel backward one square, leaving Karel facing in the same direction.

Control Statements

- In addition to allowing you to define new methods, Karel also includes three statement forms that allow you to change the order in which statements are executed. Such statements are called ***control statements***.
- The control statements available in Karel are:
  - The **`for`** statement, which is used to repeat a set of statements a predetermined number of times.
  - The **`while`** statement, which repeats a set of statements as long as some condition holds.
  - The **`if`** statement, which applies a conditional test to determine whether a set of statements should be executed at all.
  - The **`if-else`** statement, which uses a conditional test to choose between two possible actions.

The `for` Statement

- In Karel, the **`for`** statement has the following general form:

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

- As with most control statements, the **`for`** statement pattern consists of two parts:
  - The ***header line***, which specifies the number of repetitions
  - The ***body***, which is the set of statements affected by the **`for`**
- Note that most of the header line appears in boldface, which means that it is a fixed part of the **`for`** statement pattern. The only thing you are allowed to change is the number of repetitions, which is indicated by the placeholder *count*.

Using the `for` Statement

- You can use **`for`** to redefine **`turnRight`** as follows:

```
private void turnRight() {
   for (int i = 0; i < 3; i++) {
      turnLeft();
   }
}
```

- The following method creates a square of four beepers, leaving Karel in its original position:

```
private void makeBeeperSquare() {
   for (int i = 0; i < 4; i++) {
      putBeeper();
      move();
      turnLeft();
   }
}
```

Conditions in Karel

- Karel can test the following conditions:

positive condition	*negative condition*
`frontIsClear()`	`frontIsBlocked()`
`leftIsClear()`	`leftIsBlocked()`
`rightIsClear()`	`rightIsBlocked()`
`beepersPresent()`	`noBeepersPresent()`
`beepersInBag()`	`noBeepersInBag()`
`facingNorth()`	`notFacingNorth()`
`facingEast()`	`notFacingEast()`
`facingSouth()`	`notFacingSouth()`
`facingWest()`	`notFacingWest()`

The `while` Statement

- The general form of the **`while`** statement looks like this:

```
while (condition) {
   statements to be repeated
}
```

- The simplest example of the **`while`** statement is the method **`moveToWall`**, which comes in handy in lots of programs:

```
private void moveToWall() {
   while (frontIsClear()) {
      move();
   }
}
```

The `if` and `if-else` Statements

- The **`if`** statement in Karel comes in two forms:
  - A simple **`if`** statement for situations in which you may or may not want to perform an action:

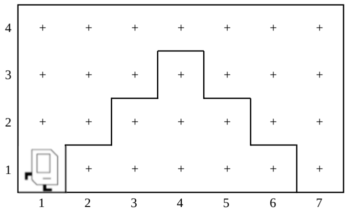
```
if (condition) {
   statements to be executed if the condition is true
}
```

  - An **`if-else`** statement for situations in which you must choose between two different actions:

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

Climbing Mountains

- For the rest of today, we'll explore the use of methods and control statements in the context of teaching Karel to climb "stair-step" mountains that look something like this:



- The initial version will work only in this world, but later examples will be able to climb mountains of any height.