Chris Piech
CS 106A

Handout #8
January 13, 2017

Example Karel Problems

Portions of this handout by Eric Roberts

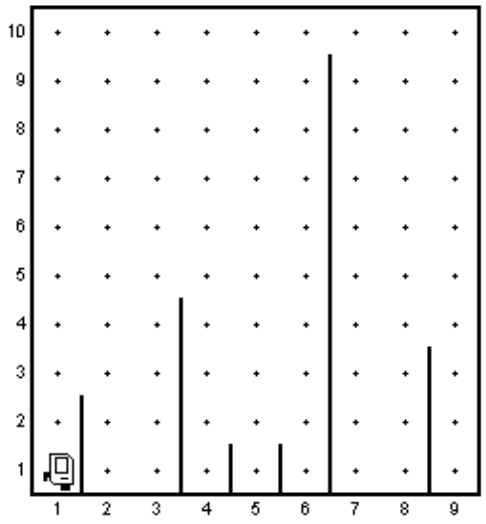
Running a steeple chase

This program that allows Karel to run a "Steeple Chase" (like a hurdles race, but with arbitrarily large hurdles) where:

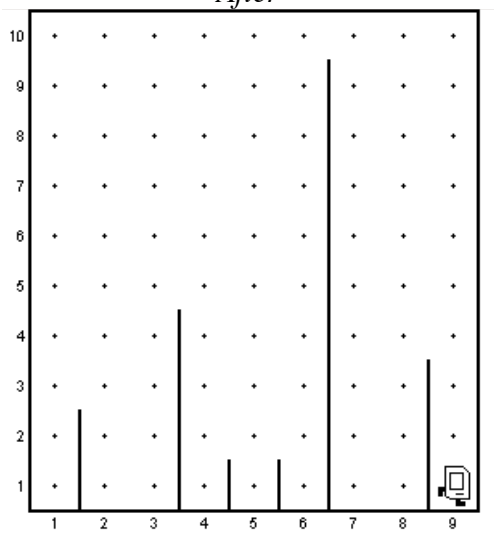
- Karel starts at position (1, 1), facing East.
- The steeple chase is guaranteed to be 9 avenues long.
- There can be arbitrarily many hurdles that can be of arbitrary size, located between any two avenues in the world.
- Karel should "jump" each hurdle one at a time.

For example, if you were to execute the **`SteepleChase`** program, you would see something like the following before-and-after diagram:

Before



After



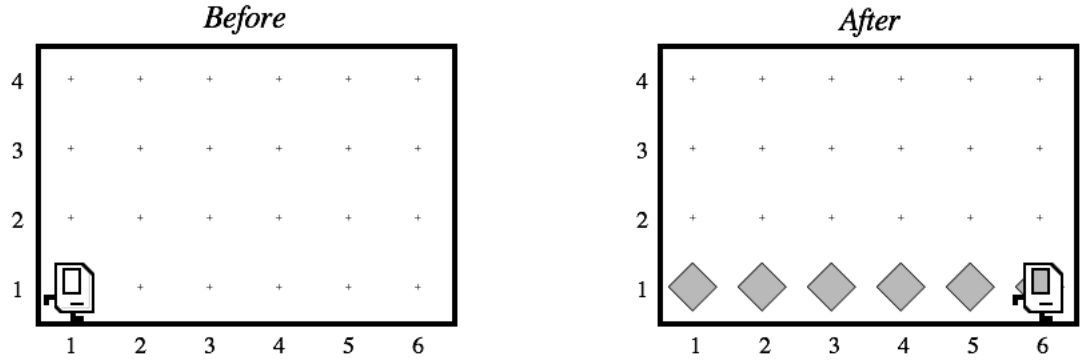
Below is the program listing of the **`SteepleChase`** program, which provides an example of various control structures in Karel, program decomposition, and comments (including the specification of pre- and post-conditions).

```
/*
 * File: SteepleChase.java
 * -----------------------
 * Karel runs a steeple chase the is 9 avenues long.
 * Hurdles are of arbitrary height and placement.
 */

import stanford.karel.*;

public class SteepleChase extends SuperKarel {

      /*
       * To run a race that is 9 avenues long, we need to move
       * forward or jump hurdles 8 times.
       */
      public void run() {
            for (int i = 0; i < 8; i++) {
                  if (frontIsClear()) {
                        move();
                  } else {
                        jumpHurdle();
                  }
            }
      }


      /*
       * Pre-condition:  Facing East at bottom of hurdle
       * Post-condition: Facing East at bottom in next avenue
       *                 after hurdle
       */
      private void jumpHurdle() {
            ascendHurdle();
            move();
            descendHurdle();
      }


      /*
       * Pre-condition:  Facing East at bottom of hurdle
       * Post-condition: Facing East immediately above hurdle
       */
      private void ascendHurdle() {
            turnLeft();
            while (rightIsBlocked()) {
                  move();
            }
            turnRight();
      }
```

```
	/*
	 * Pre-condition:  Facing East above and immediately after hurdle
	 * Post-condition: Facing East at bottom of hurdle
	 */
	private void descendHurdle() {
		turnRight();
		moveToWall();
		turnLeft();
	}


	/*
	 * Pre-condition:  none
	 * Post-condition: Facing first wall in whichever direction
	 * Karel was facing previously
	 */
	private void moveToWall() {
		while (frontIsClear()) {
			move();
		}
	}
}
```

Creating a line of beepers

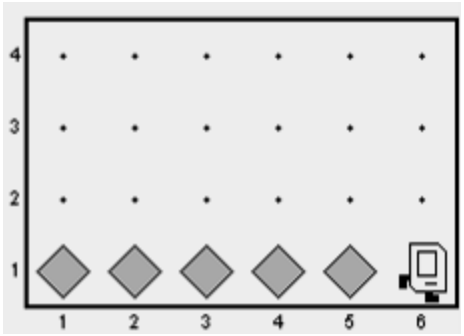
Consider the problem of writing a method **`createBeeperLine`**, which creates a line of beepers beginning at Karel's current corner and proceeding forward to the next wall. For example, if you were to execute **`createBeeperLine`** in an empty world, you would see something like the following before-and-after diagram:



The problem is slightly harder than it looks. Unless you think carefully about the problem, it is easy to find yourself making a common programming error that keeps the program from working as you'd like. For example, we might initially be inclined to solve the problem as follows:

```
	private void createBeeperLine() {
		while (frontIsClear()) {
			putBeeper();
			move();
		}
	}
```

The problem here (as indicated by the "bug" picture next to the code), is that Karel will not place a beeper on the last corner he encounters. When he reaches the final corner of the row he's in, his front will no longer be clear, so the **`while`** loop will immediately exit before a beeper is placed on that final corner. This is perhaps easier to see when looking at the Karel's world after he executes the code above:



Note that in the picture above, there is no beeper on the corner that Karel is standing on. To solve this example of a "fence-post" problem (such problems are further discussed in the Karel course reader), we must make one more **`putBeeper()`** method call than calls to **`move()`**, as shown below:

```
private void createBeeperLine() {
      while (frontIsClear()) {
            putBeeper();
            move();
      }
      putBeeper();
}
```

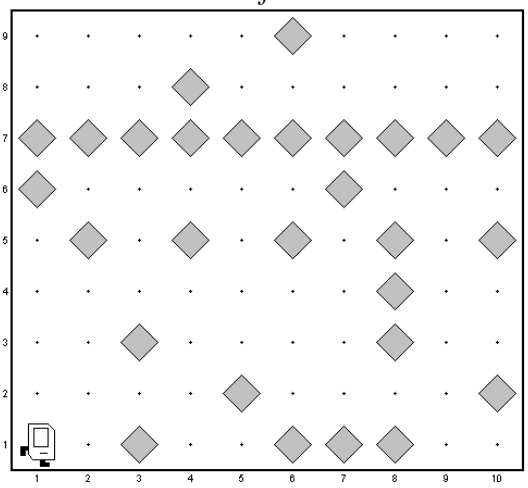
Cleaning up scattered beepers

Unfortunately, sometimes Karel's world gets a little messy, with beepers strewn around at various corners. We want to help Karel clean up his world by writing a program that has Karel go through the world and pick up any beepers that may be scattered about. We assume that:

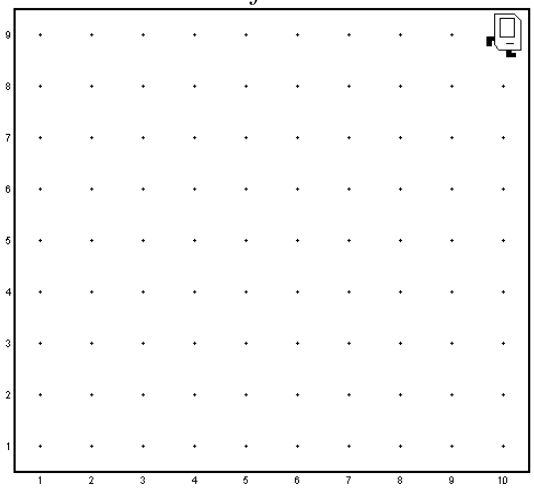
- Karel starts at corner (1, 1) facing East
- Each corner of Karel's world may either be empty or contain at most one beeper, and when Karel is done there should be no more beepers on any corner
- Karel can finish his task at any location and orientation

If you were to execute your program, you would see something like the following before-and-after diagram on the next page.

Before



After



Below is the program listing of the **`CleanupKarel`** program.

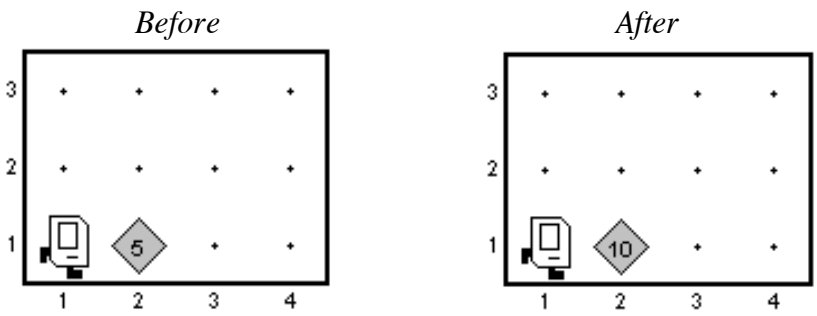
```
/*
 * File: CleanupKarel.java
 * -----------------------
 * Karel starts at (1, 1) facing East and cleans up any
 * beepers scattered throughout his world.
 */

import stanford.karel.*;

public class CleanupKarel extends SuperKarel {

    /* Cleans up a field of beepers, one row at a time */
    public void run() {
      cleanRow();
      while (leftIsClear()) {
            repositionForRowToWest();
            cleanRow();
            if (rightIsClear()) {
                  repositionForRowToEast();
                  cleanRow();
            } else {
            /*
             * In rows with an even number of streets, we want
             * Karel's left to be blocked after he cleans the last
             * row, so we turn him to the appropriate orientation.
             */
                  turnAround();
            }
      }
    }
```

```
	/* Cleans up a row in whichever direction Karel is facing */
	private void cleanRow() {
		if (beepersPresent()) {
			pickBeeper();
		}
		while (frontIsClear()) {
			move();
			if (beepersPresent()) {
				pickBeeper();
			}
		}
	}


	/* Reposition Karel at far East wall to face West on next row */
	private void repositionForRowToWest() {
		turnLeft();
		move();
		turnLeft();
	}


	/* Reposition Karel at far West wall to face East on next row */
	private void repositionForRowToEast() {
		turnRight();
		move();
		turnRight();
	}

}
```

Doubling the number of beepers in a pile

Interestingly enough, Karel can also do various kinds of math. For example, we can ask Karel to double the number of beepers on some corner in the world. We'd like to write a program where there is a pile of some (finite) number of beepers on the corner directly in front of Karel. We want Karel to double the number of beepers in that pile and return to his original location and orientation. You can assume that Karel begins with an infinite number of beepers in his beeper bag.

For example, if you were to execute your program in the sample *Before* world shown below, you would expect to see the corresponding *After* world (note that the number of beepers on corner (2, 1) has been doubled):



Below is the program listing of the **DoubleBeepers** program.

```
/*
 * File: DoubleBeepers.java
 * ------------------------
 * Karel doubles the number of beepers on the corner directly
 * in front of him in the world.  He then returns to his
 * original position/orientation.
 */

import stanford.karel.SuperKarel;

public class DoubleBeepers extends SuperKarel {

      public void run() {
            move();
            doubleBeepersInPile();
            moveBackward();
      }


      /*
       * For every beeper on the current corner, Karel places
       * two beepers on the corner immediately ahead of him.
       */
      private void doubleBeepersInPile() {
            while (beepersPresent()) {
                  pickBeeper();
                  putTwoBeepersNextDoor();
            }
            movePileNextDoorBack();
      }


      /*
       * Place two beepers on corner one avenue ahead of Karel
       * and move back to starting position/orientation
       */
      private void putTwoBeepersNextDoor() {
            move();
            putBeeper();
            putBeeper();
            moveBackward();
      }
```

```
	/*
	 * Move all the beepers on the corner in front of Karel
	 * the the corner Karel is currently on.
	 */
	private void movePileNextDoorBack() {
		move();
		while (beepersPresent()) {
			moveOneBeeperBack();
		}
		moveBackward();
	}



	/*
	 * Move one beeper from the current corner back one avenue
	 * and return to the original position/orientation.
	 */
	private void moveOneBeeperBack() {
		pickBeeper();
		moveBackward();
		putBeeper();
		move();
	}


	/*
	 * Move Karel back one avenue, but have the same
	 * final orientation.
	 */
	private void moveBackward() {
		turnAround();
		move();
		turnAround();
	}

}
```