CS 106A, Lecture 3
Problem-Solving with Karel

reading:

Karel the Robot Learns Java, Chapters 3-6

Karel condition methods

- Karel has some commands that are not meant to be complete statements, but rather are used to ask questions:

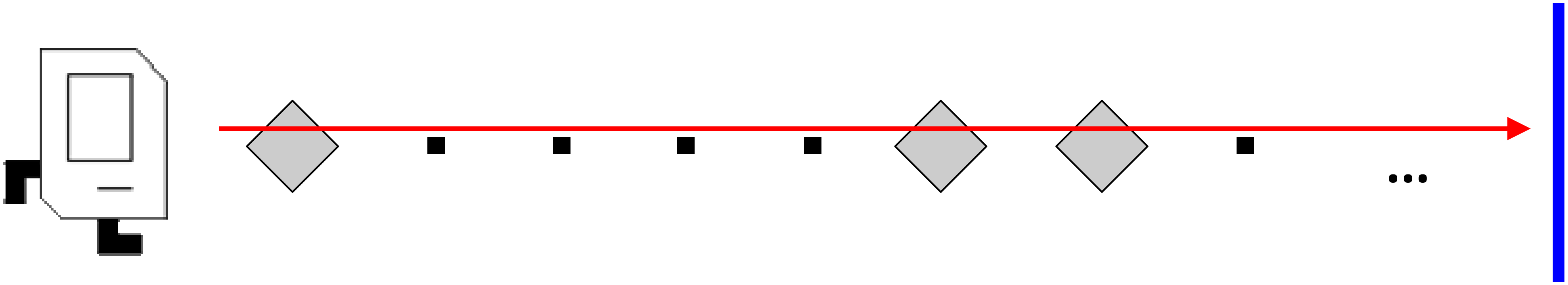
Test	***Opposite***	***What it checks***
`frontIsClear()`	**`frontIsBlocked()`**	Is there a wall in front of Karel?
`leftIsClear()`	**`leftIsBlocked()`**	Is there a wall to Karel's left?
`rightIsClear()`	**`rightIsBlocked()`**	Is there a wall to Karel's right?
`beepersPresent()`	**`noBeepersPresent()`**	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?

*This is **Table 1** on page 18 of the Karel course reader.*

Exercise: Sweeper

Sweeper

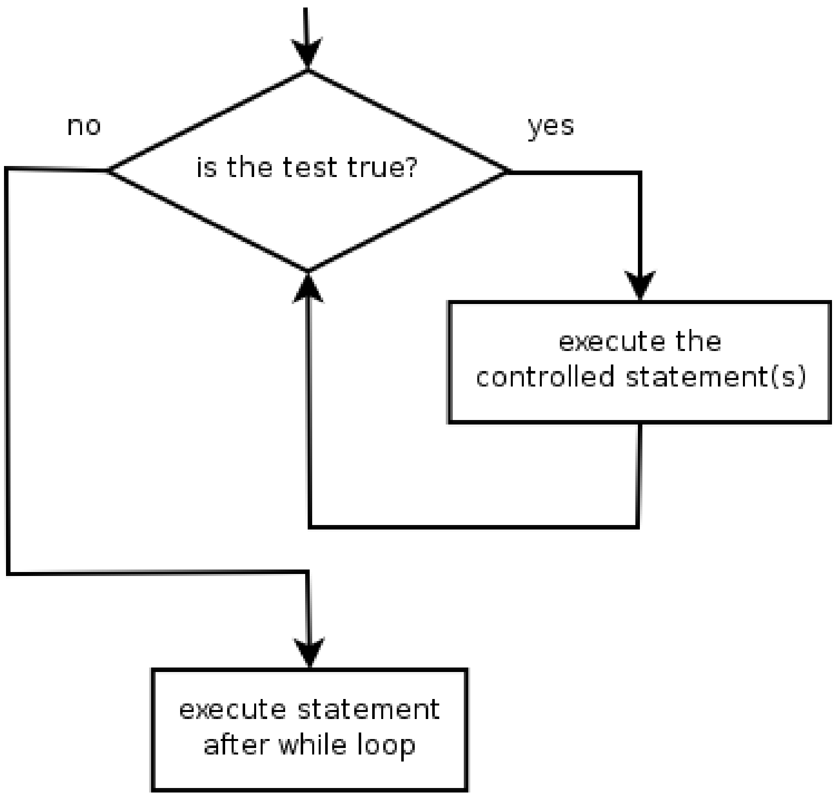
- Recall, last lecture we wrote a "**sweeper**" that picks up all beepers in front of Karel in a straight line.
- Suppose we want our sweeper to walk Karel all the way to the edge of the world (or the nearest wall), regardless of the world's size.
  - What should we set our `for` loop's ***max*** to?
  - Is a `for` loop really the right tool for solving this problem?



The while loop

Repeatedly executes its body as long as a logical test is true

```
while (test) {
    statements;
}
```



- Example:

```
// drop all of my beepers
while (beepersInBag()) {
    putBeeper();
}
```

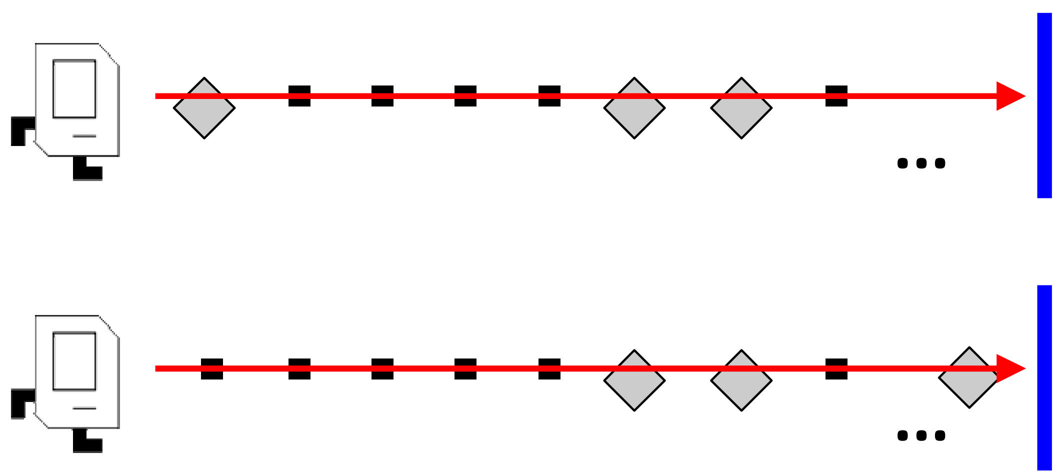
  – `while` is different from `if`.
  `if` checks the test just once.
  `while` checks it repeatedly until it fails.

Question

```
// solution 1
while (frontIsClear()) {
    if (beepersPresent()) {
        pickBeeper();
    }
    move();
}
```

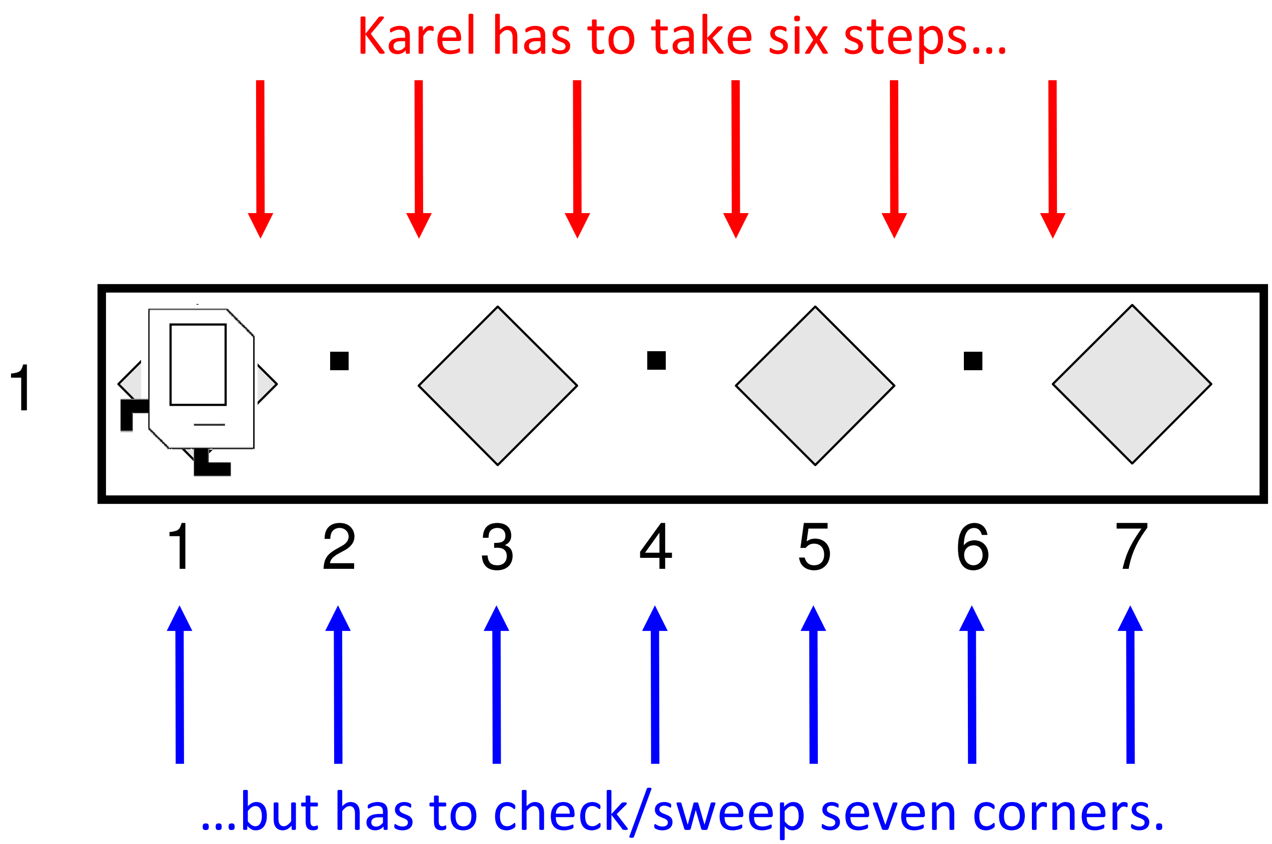
```
// solution 2
while (frontIsClear()) {
    move();
    if (beepersPresent()) {
        pickBeeper();
    }
}
```

- **Q:** Which solution works for all cases?
    - **A.** Solution 1
    - **B.** Solution 2
    - **C.** Both
    - **D.** Neither



A tricky bug

- Our code fails when there is a beeper at the final corner.
  - Changing the order of statements is likely to make it fail when there is a beeper at the *first* corner. It must work for both.



Fencepost problem

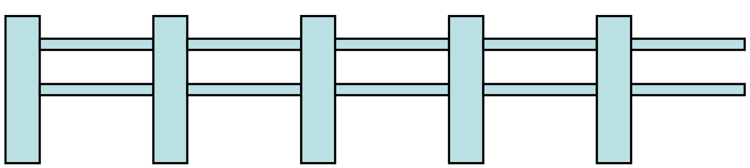
- **fencepost problem**: One with repeated statements, where some statements should be repeated *n* times and some *n*-1 times.
  - Like creating a fence with 5 posts and 4 wires between the posts.

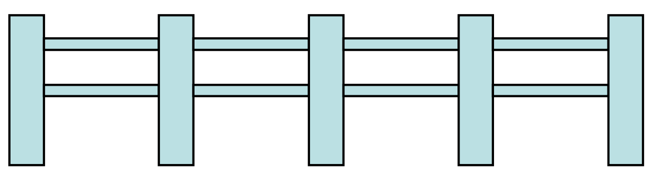
Incorrect:

```
loop {
    place a post.
    place some wire.
}
```

Correct:

```
place a post.
loop {
    place some wire.
    place a post.
}
```





Fencepost loop

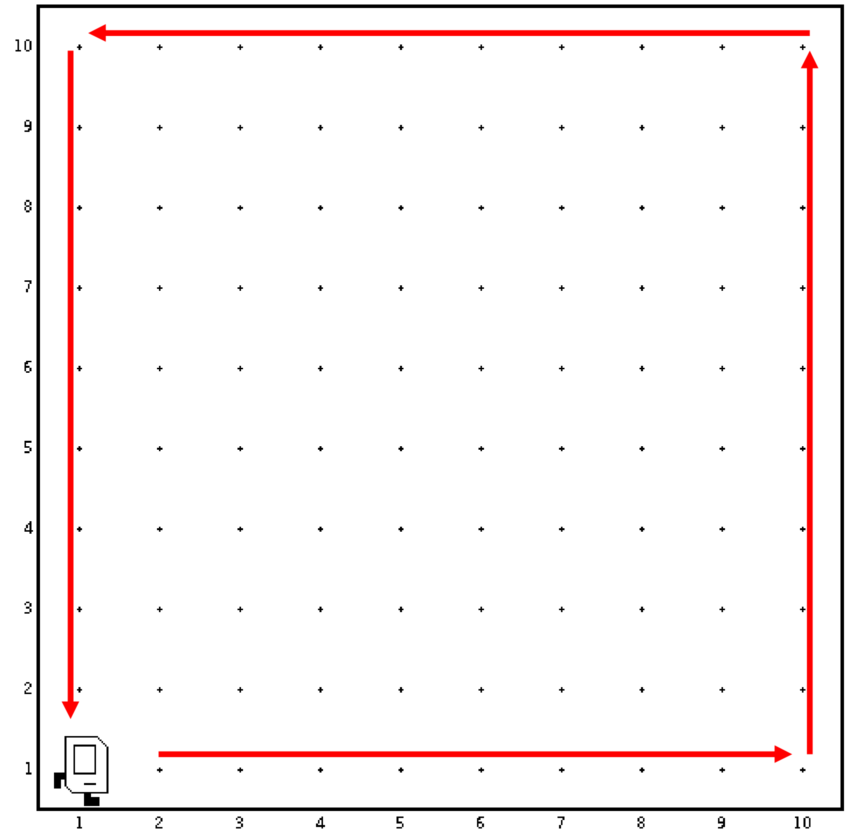
- To solve a fencepost problem, perform the task that needs to happen *n* times (the "post") once outside the loop.
  - If necessary, invert the steps inside the loop ("wire", then "post").

```
public void run() {
    safePickup();                   // post
    while (frontIsClear()) {
        move();                     // wire
        safePickup();               // post
    }
}

public void safePickup() {
    if (beepersPresent()) {
        pickBeeper();
    }
}
```

Exercise: Racetrack

Racetrack

- Write a **`Racetrack`** Karel that walks around the world's perimeter one time to completion.
- Your code should work for a world of any size.



You may assume:

- Karel starts at (1,1) facing E
- the world is rectangular
- there are no walls other than the outer border

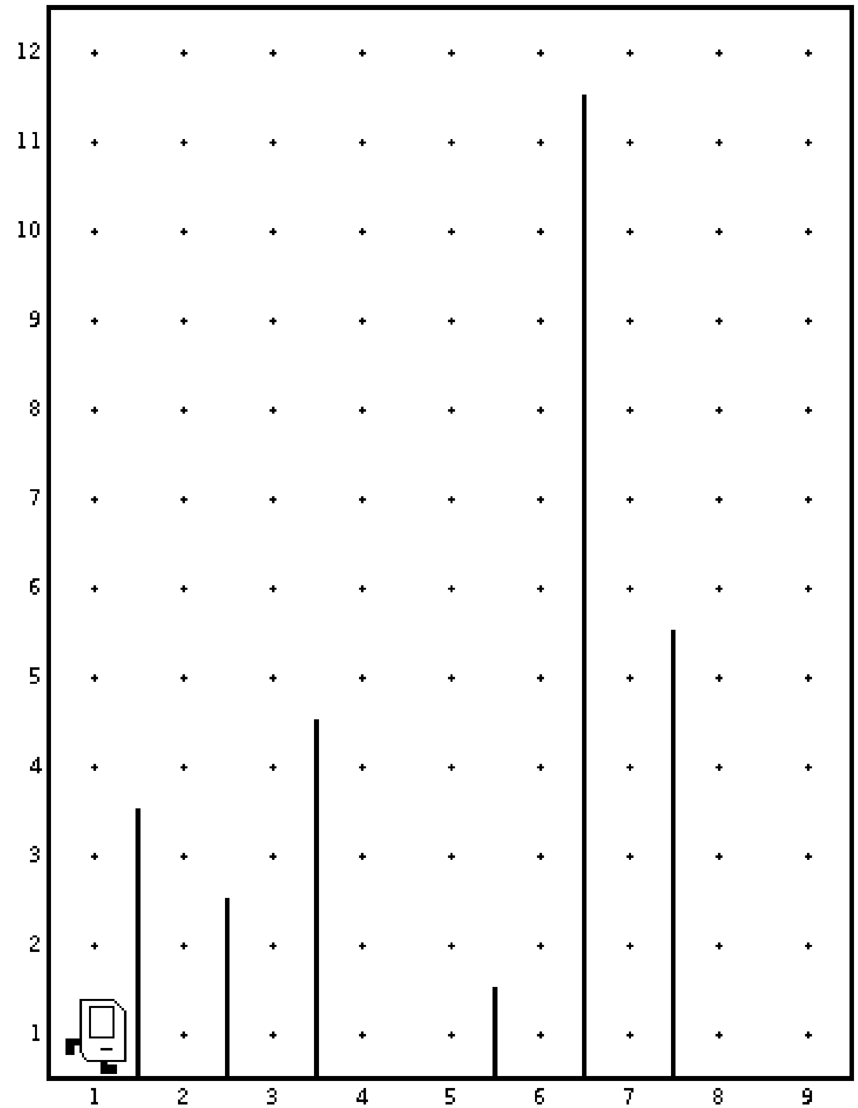
Racetrack solution

```
import stanford.karel.*;

public class Racetrack extends Karel {
    public void run() {
        for (int i = 0; i < 4; i++) {
            runOneLength();
        }
    }

    public void runOneLength() {
        while (frontIsClear()) {
            move();
        }
        turnLeft();
    }
}
```

Exercise: Hurdle jumper

HurdleJumper

- Write a **`HurdleJumper`** that can move Karel up and over 8 "hurdles" made of walls of arbitrary height.



You may assume:

- Karel starts at (1,1) facing East
- Each hurdle is a vertical wall
- The ground is "flat" otherwise
- Every hurdle can be jumped over

Hurdle jumper solution

```
public class HurdleJumper extends SuperKarel {
    public void run() {
        for (int i = 0; i < 8; i++) {
            jump();
        }
    }
    public void jump() {
        ascendHurdle();
        move();
        descendHurdle();
    }
    public void ascendHurdle() {
        turnLeft();
        while (rightIsBlocked()) {
            move();
        }
        turnRight();
    }
    public void descendHurdle() {
        turnRight();
        while (frontIsClear()) {
            move();
        }
        turnLeft();
    }
}
```

Pre/post comments

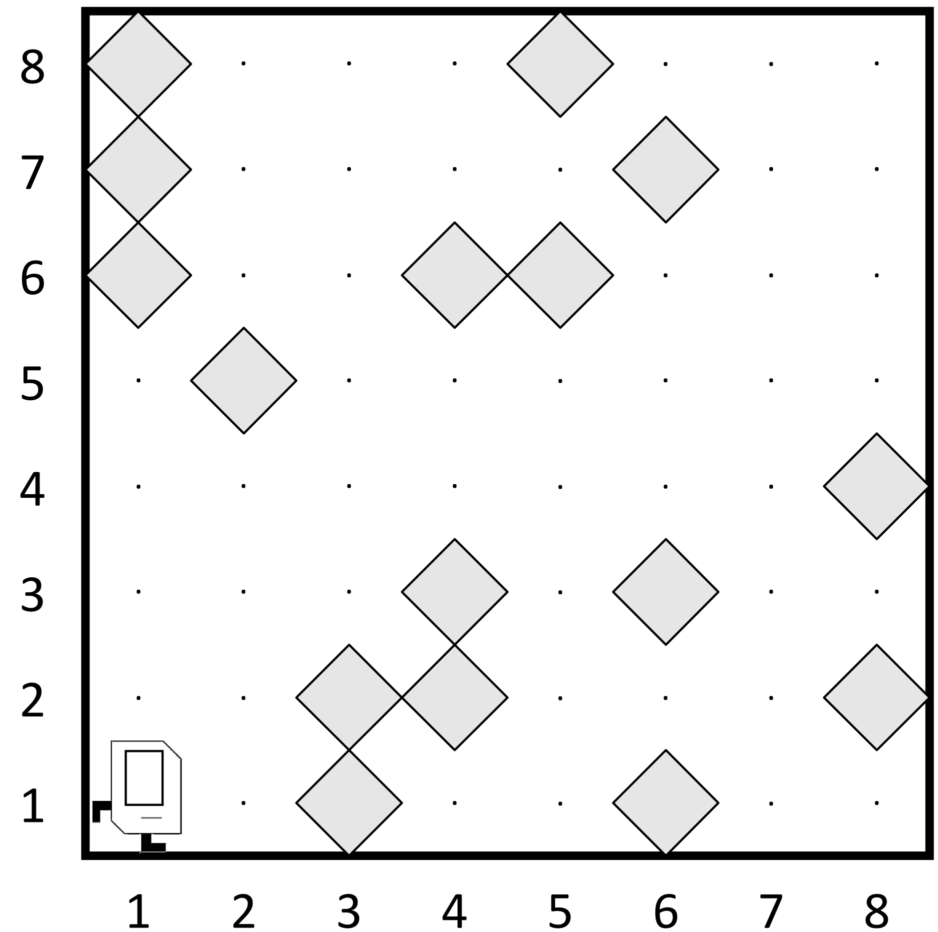
- **precondition**: Something you *assume* is true at the start of a method.
- **postcondition**: Something you *promise* is true at the end of a method.
  - pre/post conditions should be documented using comments.

```
/*
 * Jumps Karel over one hurdle of arbitrary height.
 * Pre:  Karel is facing right, next to a jumpable hurdle.
 * Post: Karel will be over the hurdle, facing right.
 */
public void jump() {
    ascendHurdle();
    move();
    descendHurdle();
}
```

  - What are pre/post conditions of other methods we have written?

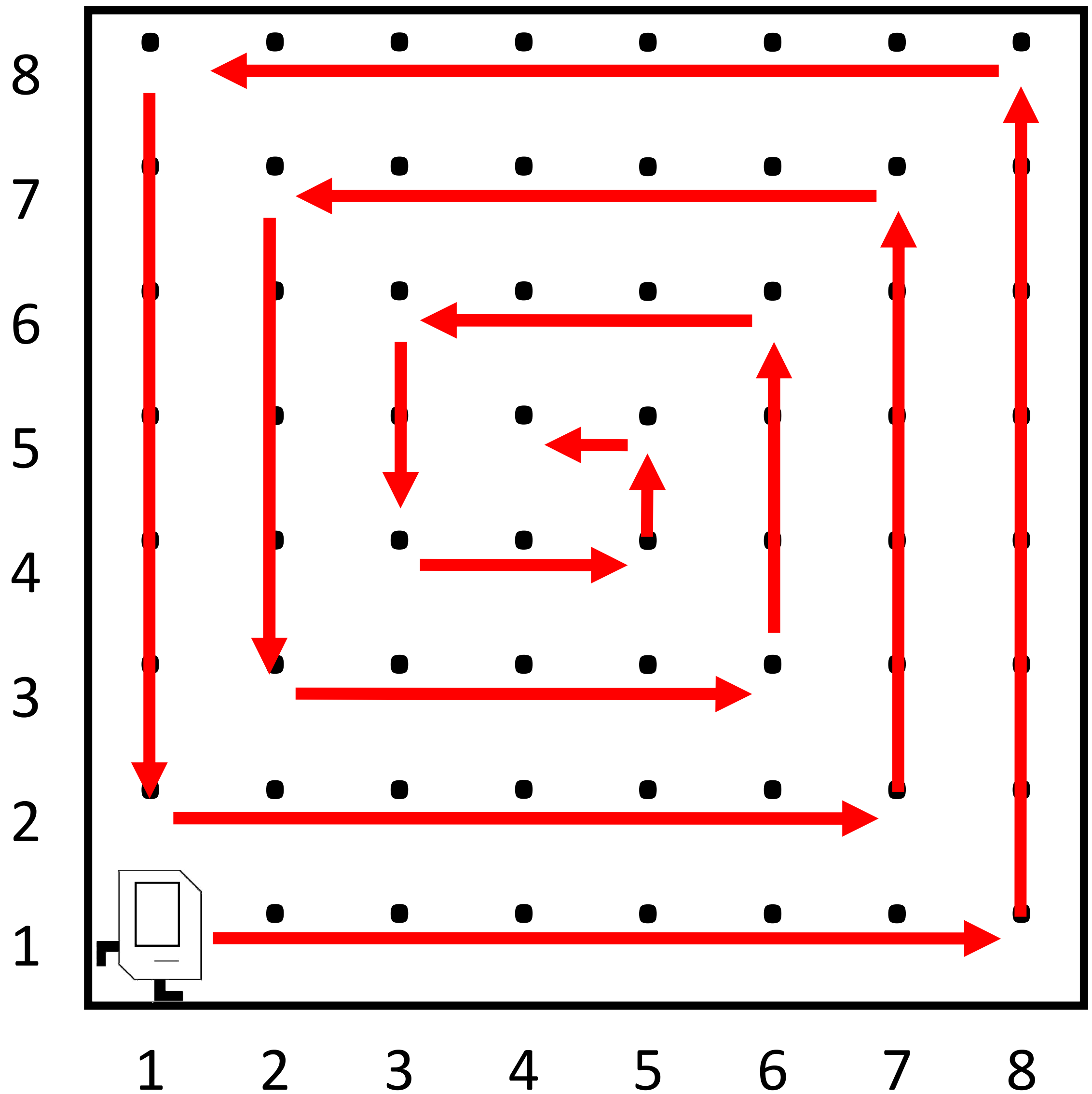
Exercise: Roomba

- Write a **Roomba** Karel that sweeps the entire board of all beepers.
  - Karel starts at (1,1) facing East.
  - The world is rectangular, and some squares contain beepers.
  - There are no interior walls.
  - When the program is done, the world should contain 0 beepers.
  - Karel's ending location does not matter.

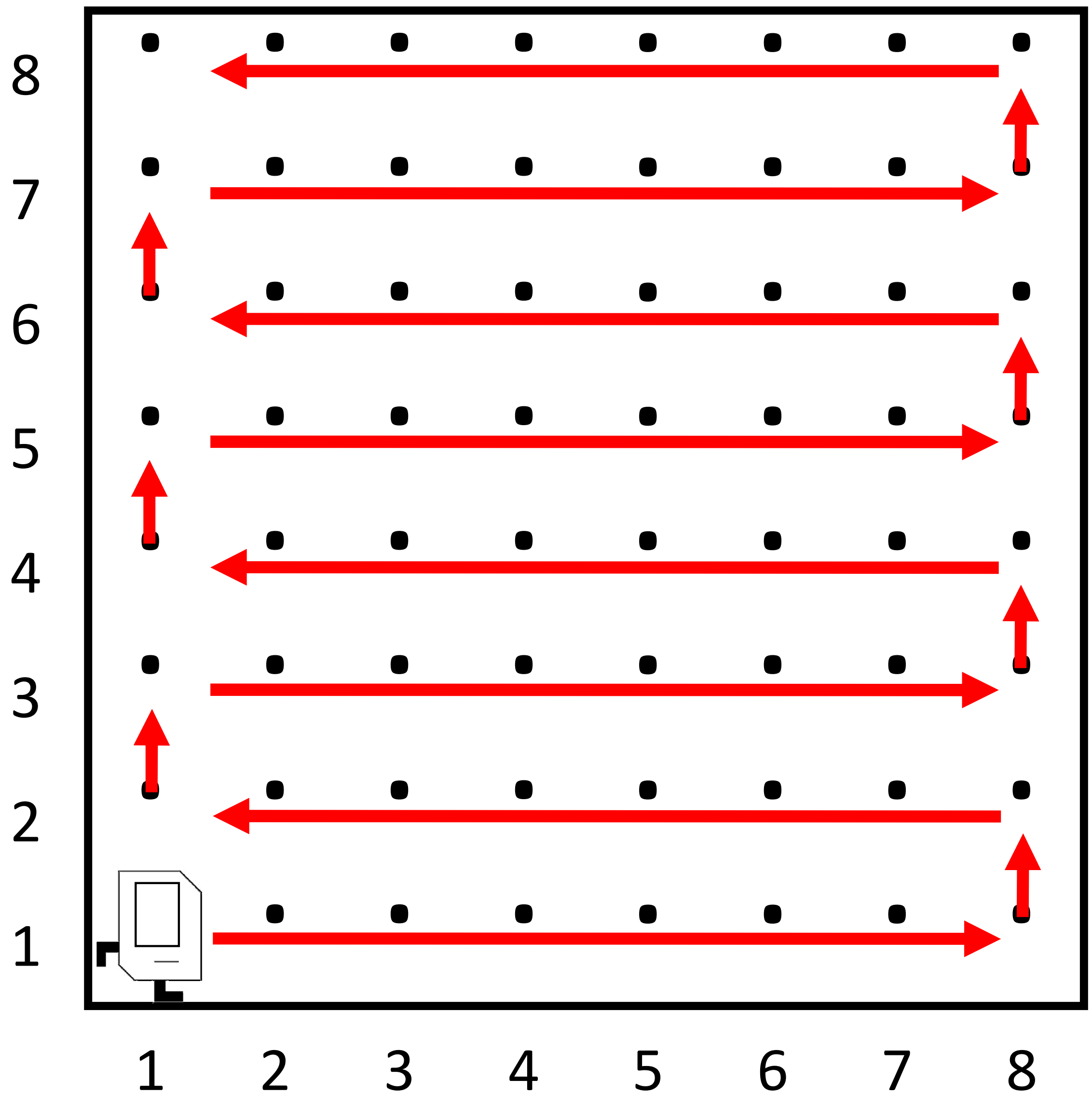


- How should we approach this tricky problem?

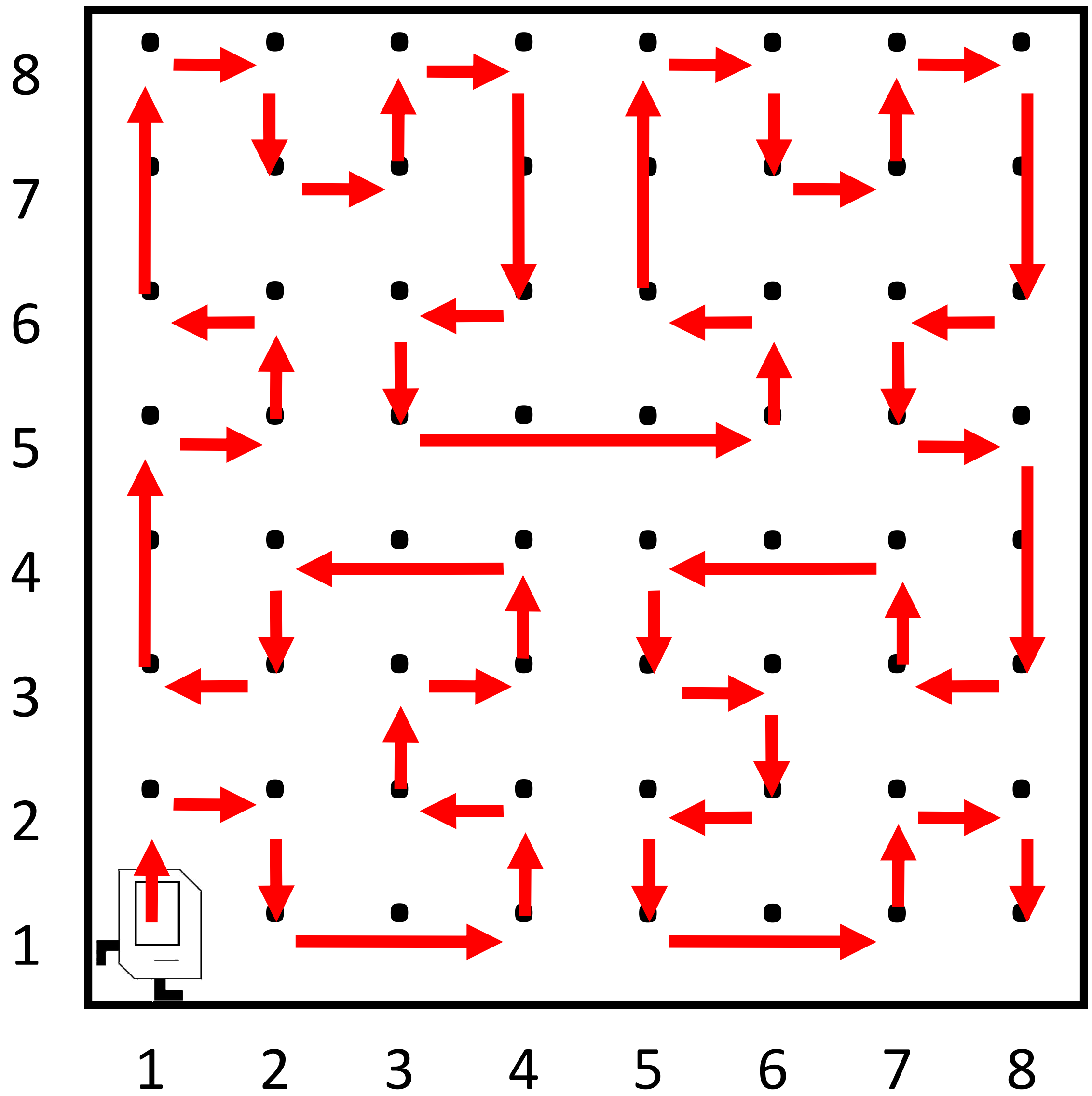
Possible algorithm 1



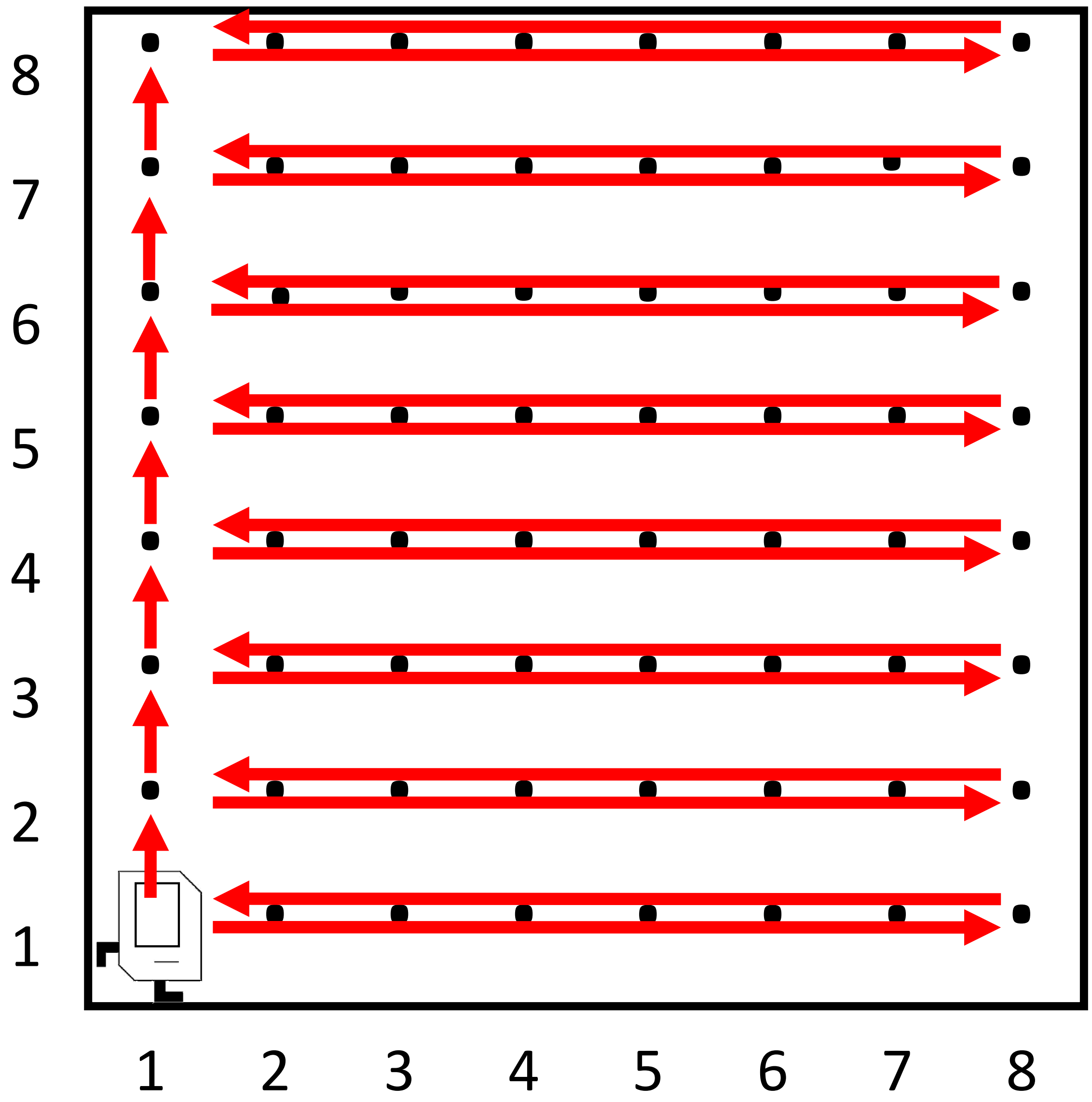
Possible algorithm 2



Possible algorithm 3



Possible algorithm 4



Roomba solution

```
import stanford.karel.*;

public class RoombaKarel extends SuperKarel {
    public void run() {
        sweep();
        while (leftIsClear()) {
            moveUp();
            sweep();
        }
    }

    public void sweep() {
        safePickup();
        while (frontIsClear()) {
            move();
            safePickup();
        }
        returnHome();
    }
    ...
```

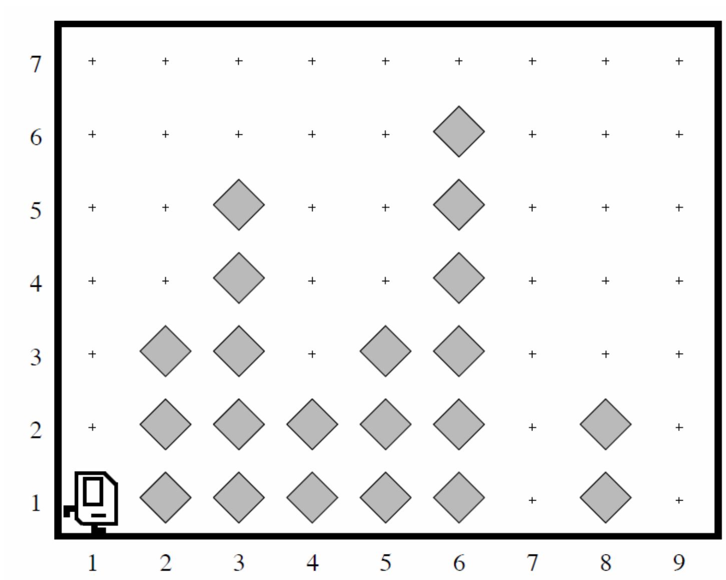
Roomba solution cont'd.

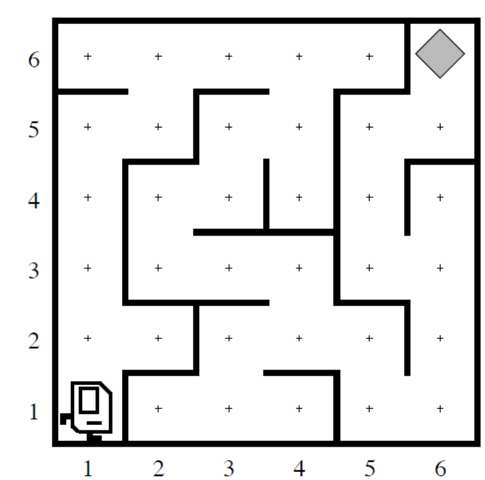
```
    ...
    public void returnHome() {
        turnAround();
        while (frontIsClear()) {
            move();
        }
        turnAround();
    }

    public void moveUp() {
        turnLeft();
        move();
        turnRight();
    }

    public void safePickup() {
        if (beepersPresent()) {
            pickBeeper();
        }
    }
}
```

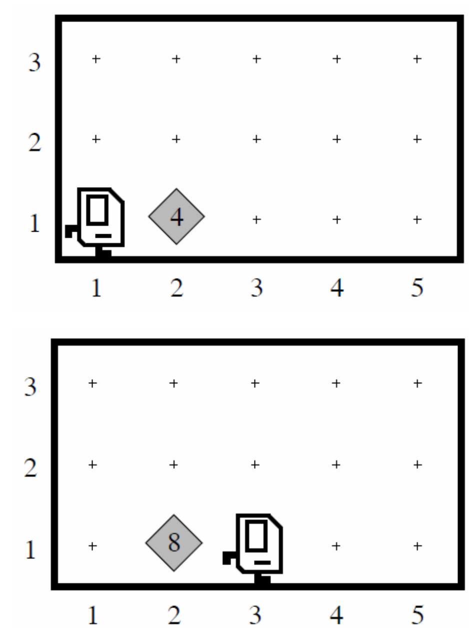
Practice problems

BeeperCollector
MazeRunner
DoubleBeepers

- Ch. 4: **`BeeperCollectingKarel`**; collect beepers from tall towers
- Ch. 5: **`MazeRunningKarel`**; find beeper in maze by "right-hand rule"
- Ch. 5: **`DoubleBeepers`**; double number of beepers in a corner (harder than it sounds!)







Private methods

- **private method:** An advanced concept. Public methods can be called by other classes/programs. Private ones cannot.
  - Not really relevant for our Karel programs, which always have 1 class.
  - The book examples always use private methods. (except `run`)
  - Your methods on HW1 can be either public or private. (Up to you)

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

private void safePickup() {
    if (beepersPresent()) {
        pickBeeper();
    }
}
```

Advanced: Colors

- The `SuperKarel` class has an additional method `paintCorner` that sets Karel's current position to be a given color.
  - Valid colors are `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, `YELLOW`, and `null` (no color)

```
import stanford.karel.*;

public class RedStripe extends SuperKarel {
    public void run() {
        while (frontIsClear()) {
            paintCorner(RED);
            move();
        }
    }
}
```