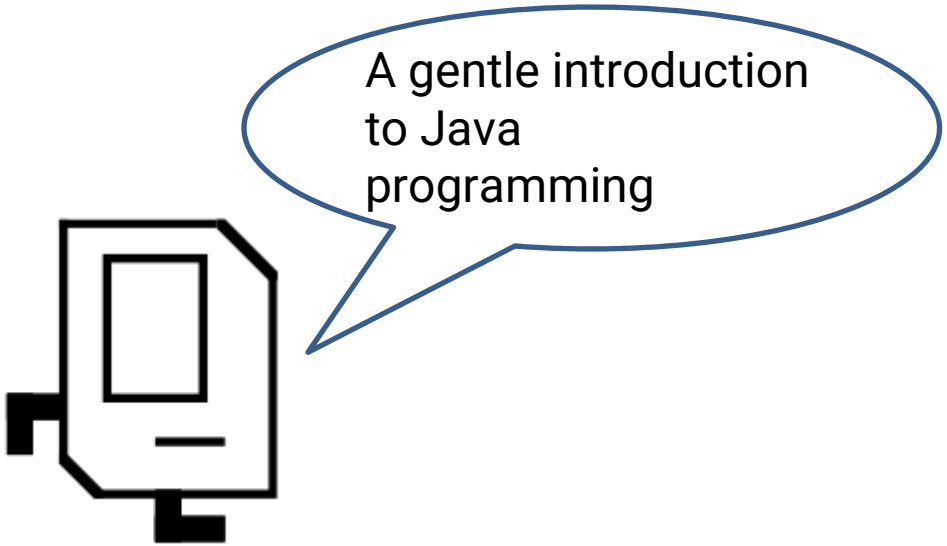
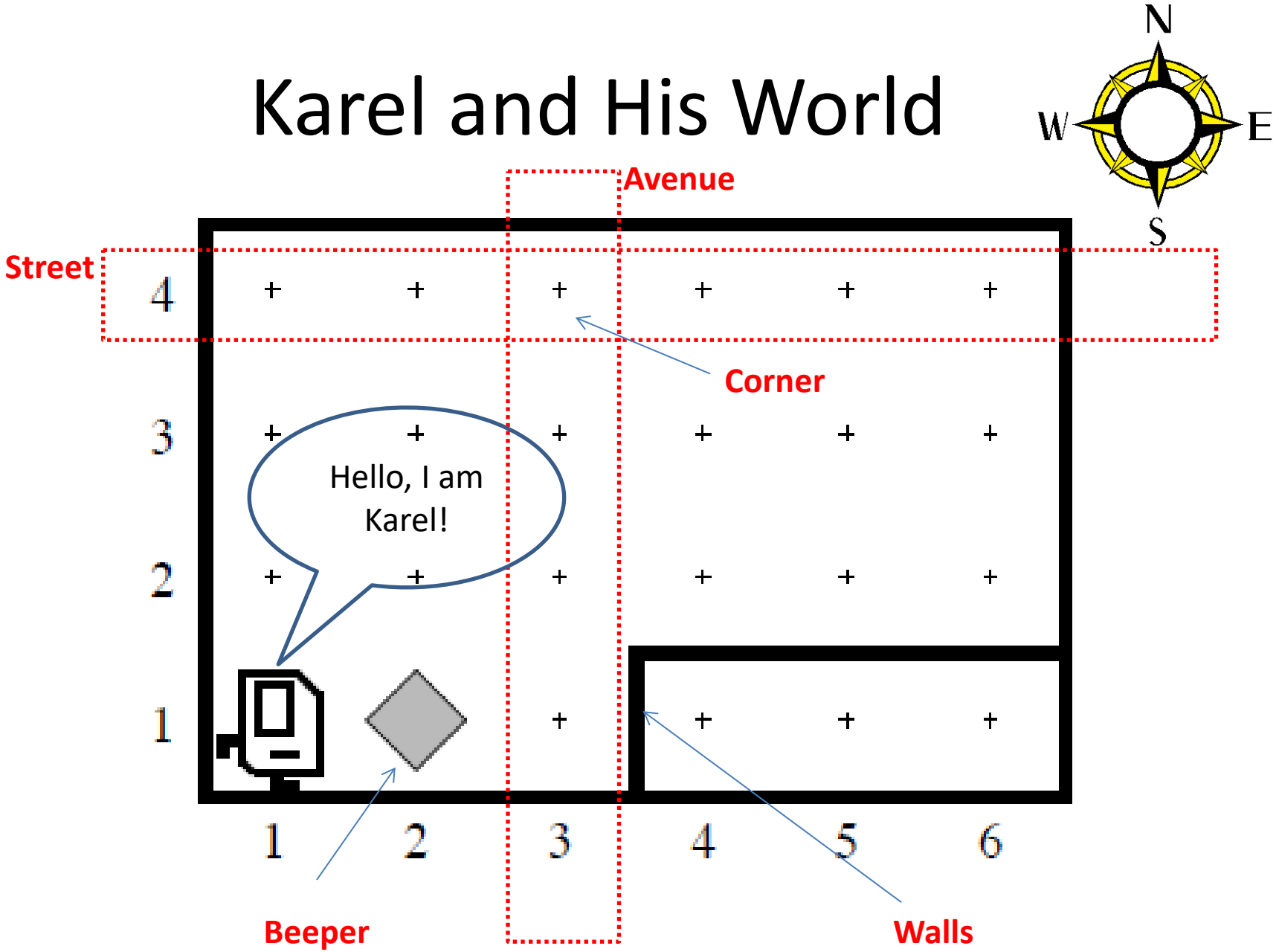
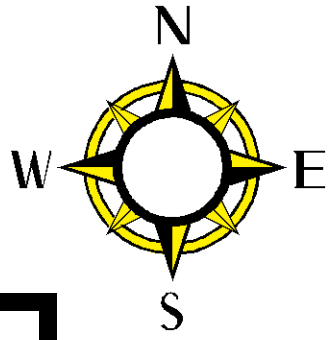
Karel the Robot

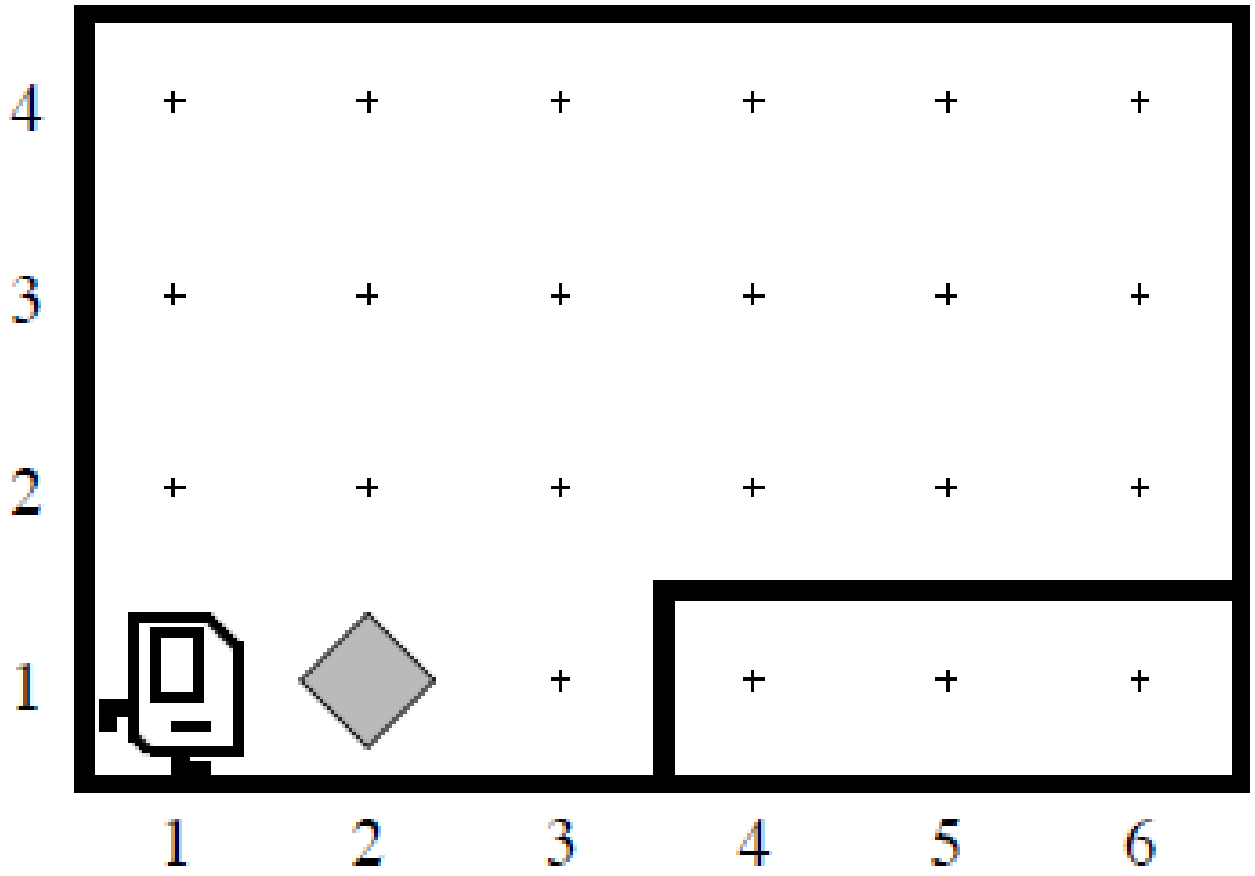


Karel and His World



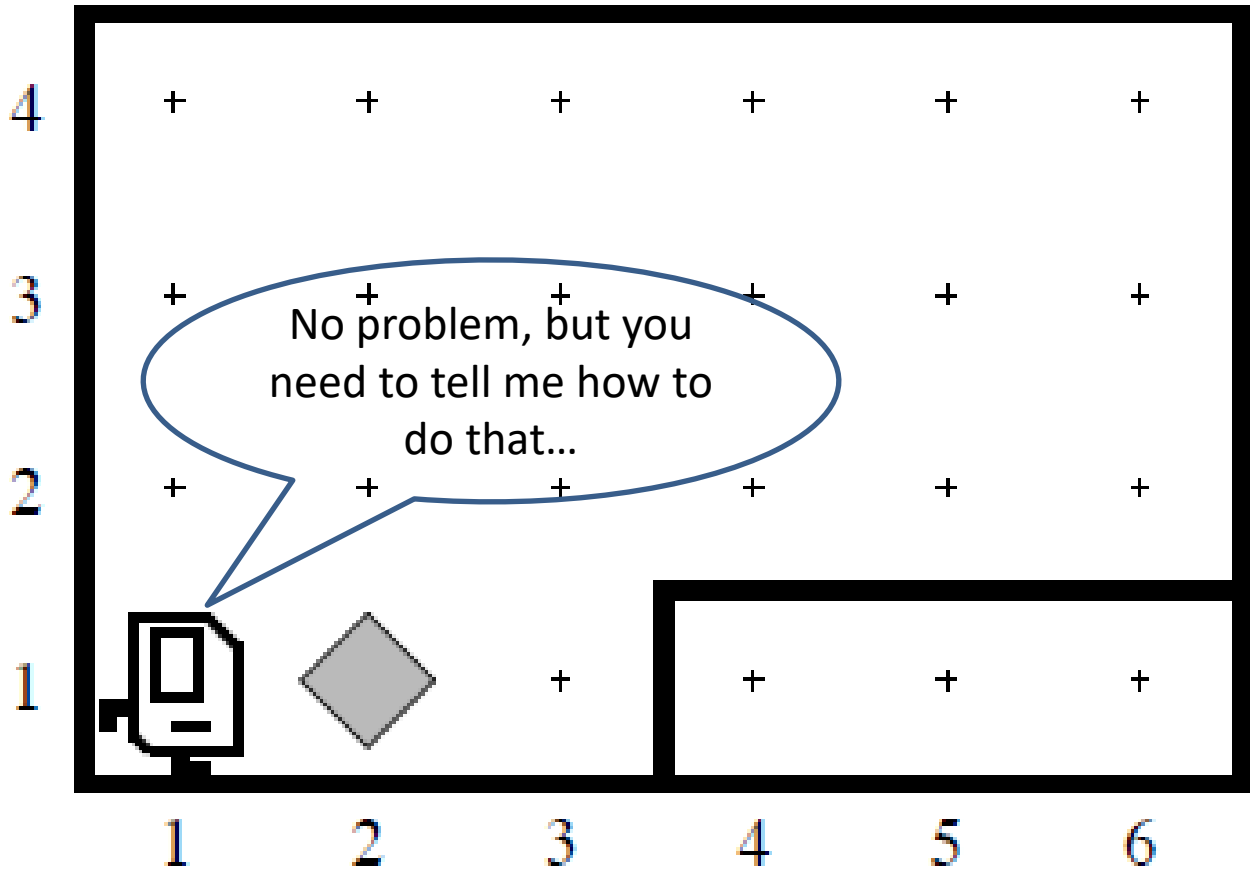
What Can Karel Do?



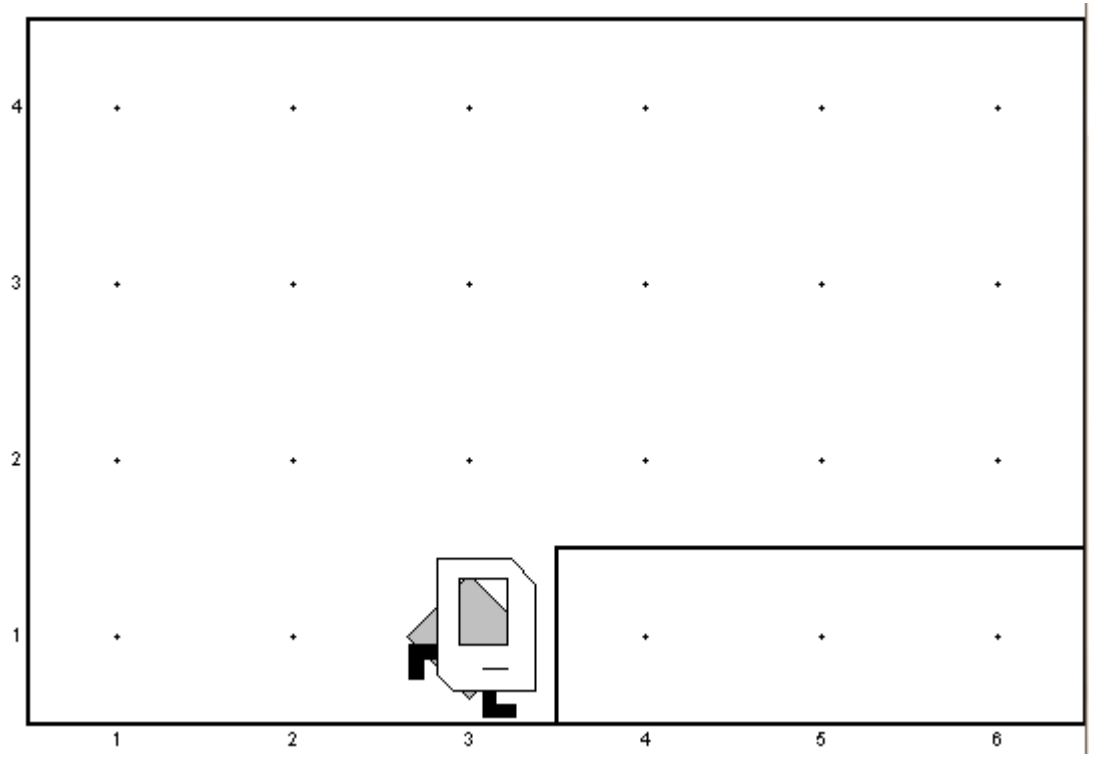


Karel can move, turnLeft, pickBeeper, putBeeper.
Karel can also sense certain things about his world.

Karel, Pick Up That Beeper!!!



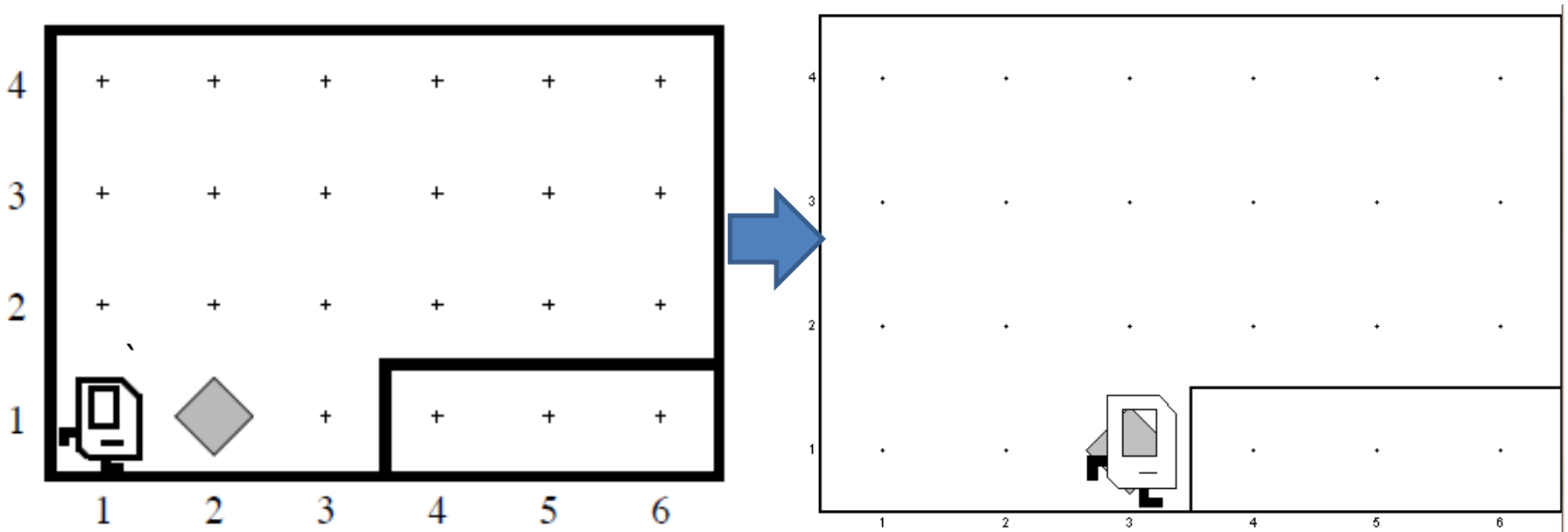
Karel, Pick Up That Beeper!!!



We want Karel to pick up this beeper, drop it off at the next corner, and end up at that corner facing to the east.

How We Might Do That With The Instructions We Have?

- move
- turnLeft
- pickBeeper
- putBeeper

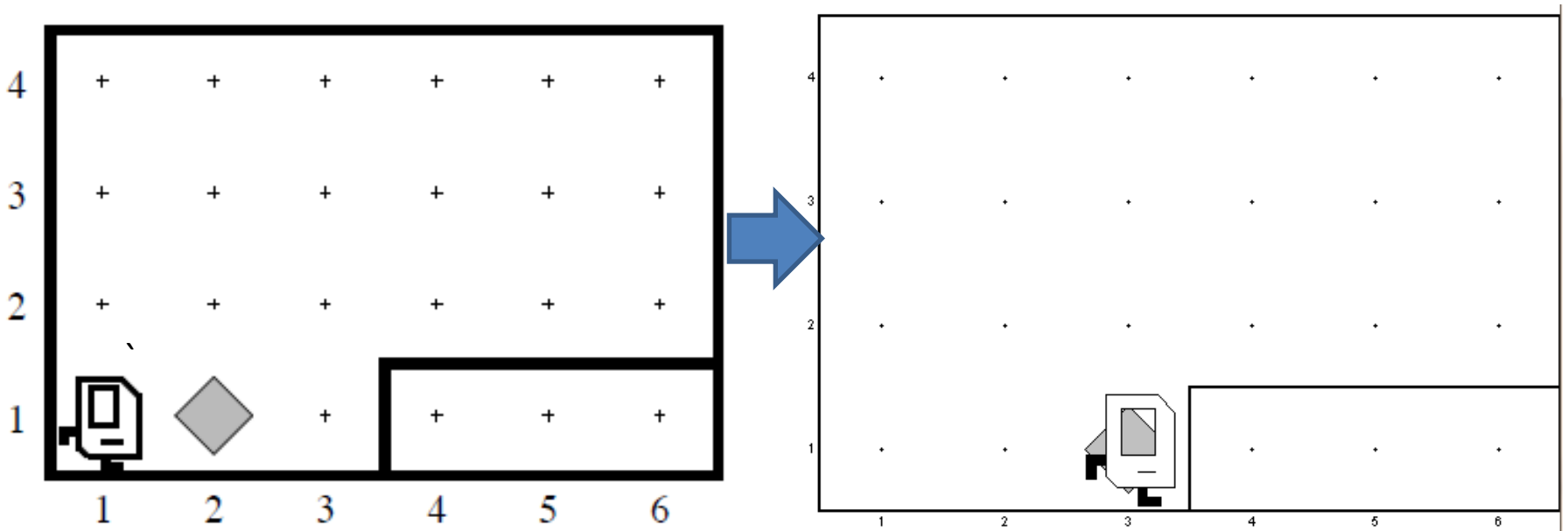


A Sequence of Commands

1. move
2. pickBeeper
3. move
4. putBeeper

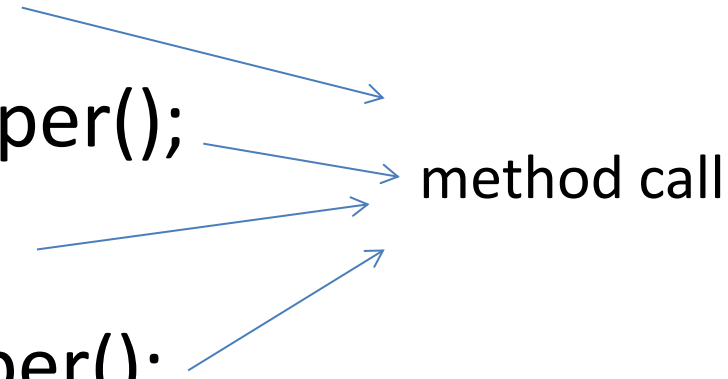
This is the algorithm of this task!!!

Is this the only way to achieve this task???



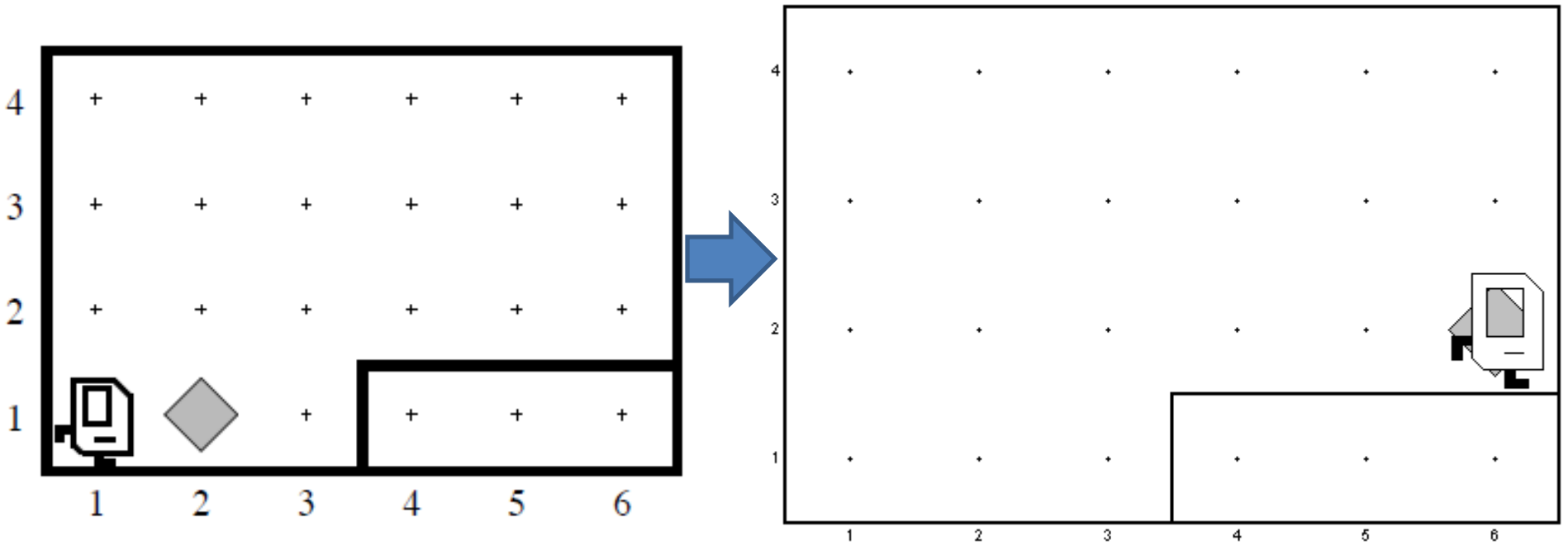
Turn Algorithm to Program

Algorithm:	Karel program(incomplete):
1. move	move();
2. pickBeeper	pickBeeper();
3. move	move();
4. putBeeper	putBeeper();



Now we get the valid commands, but we still cannot run the program.

Practice 1

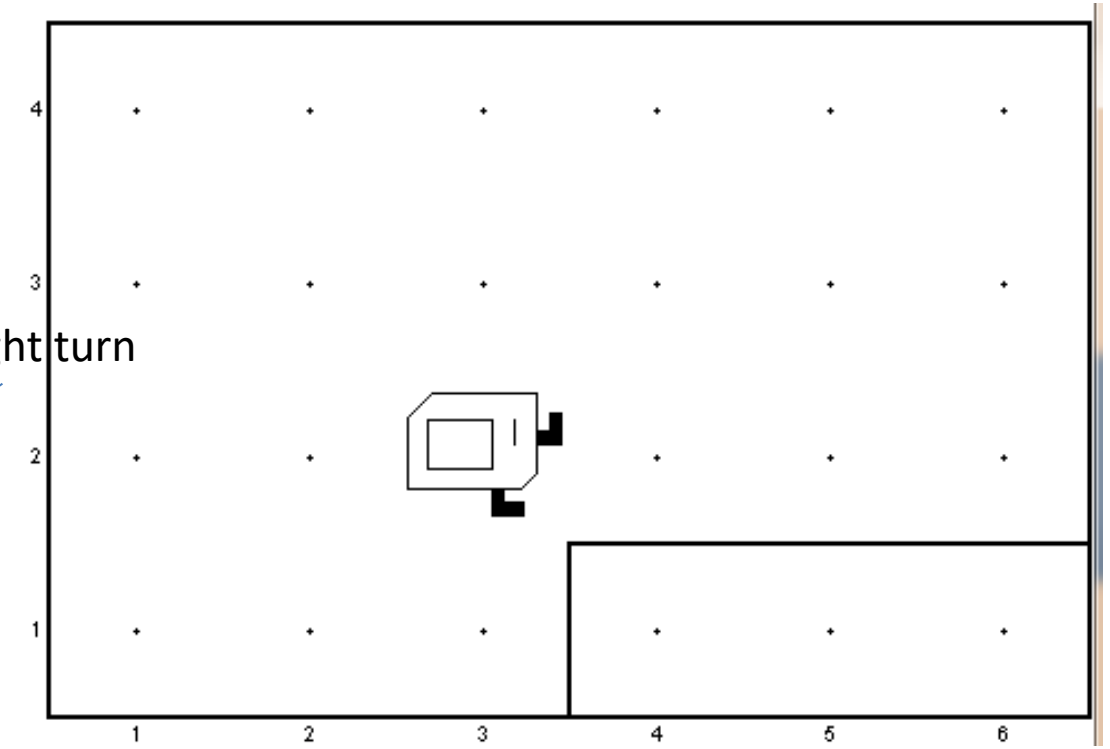


Bonus: Assuming there are infinite beepers in Karel's bag, modify your program so that Karel would put 5 beepers on the final corner.

Solutions for Practice 1

```
import stanford.karel.Karel;
public class BeeperPickingKarel extends Karel
{
    public void run()
    {
        move();
        pickBeeper();
        move();
        turnLeft();
        move();
        turnLeft();
        turnLeft();
        turnLeft();
        move();
        move();
        move();
        putBeeper();
    }

}
```

Makes a right turn



Karel's position before three turnLeft() method calls;

```java
/*
* File: BeeperPickingKarel.java
* The BeeperPickingKarel class extends the basic Karel class
* by defining a "run" method with several commands. These
* commands cause Karel to move forward one corner, pick up
* a beeper, and then "climb" the wall and put beeper at the end of the wall.
* we use the world "KarelStartWorld"!!!
*/
import stanford.karel.Karel;
public class BeeperPickingKarel extends Karel
{
    public void run()
    {
        //move forward one corner and pick up the beeper on that corner
        move();
        pickBeeper();
        //move one more corner
        move();
        //"climb" the wall
        turnLeft();
        move();
        //The three turnLefts makes a right turn!!!
        turnLeft();
        turnLeft();
        turnLeft();
        //move to the wall and drop the beeper at the corner
        move();
        move();
        move();
        putBeeper();
    }
}
```

We put comments to clarify things in the program that are not obvious!

Always bear in mind that we write programs for people to read, not just for computer to read!!!

Defining New Methods

Original program

```
import stanford.karel.Karel;
public class BeeperPickingKarel extends Karel
{
    public void run()
    {
        move();
        pickBeeper();
        move();
        turnLeft();
        move();
        turnLeft();
        turnLeft();
        turnLeft();
        move();
        move();
        move();
        putBeeper();
    }

}
```

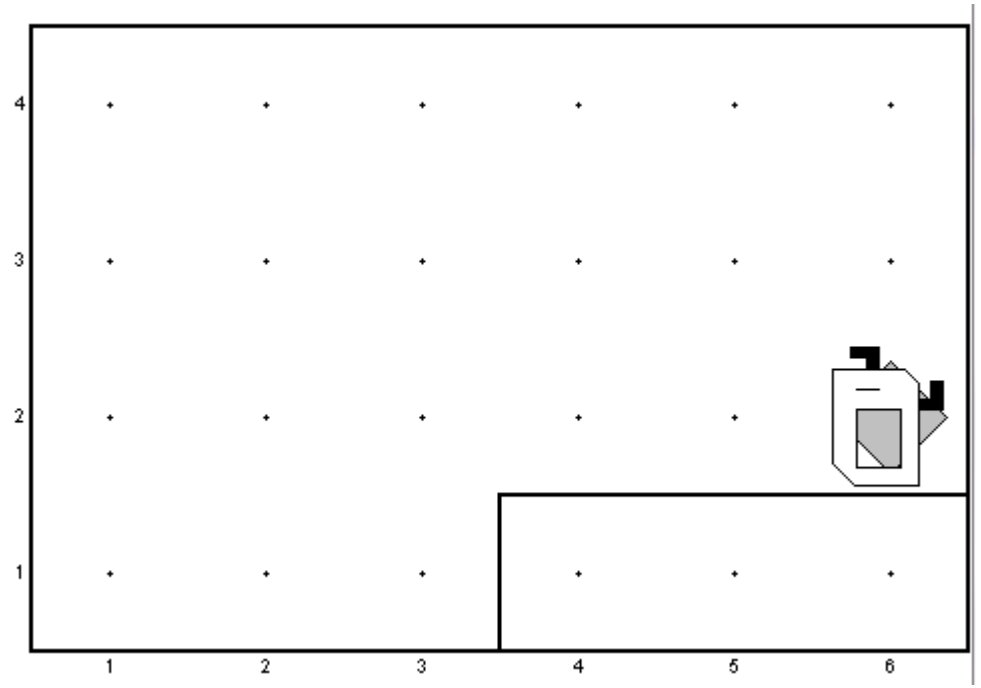
modified program

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

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

}
```

Go to the turnRight method body, run all the instructions then return to where it left off.

Practice 2

- Write a method named turnAround() in class BeeperPickingKarel, so that after Karel puts beeper down at his destination, he turns around.

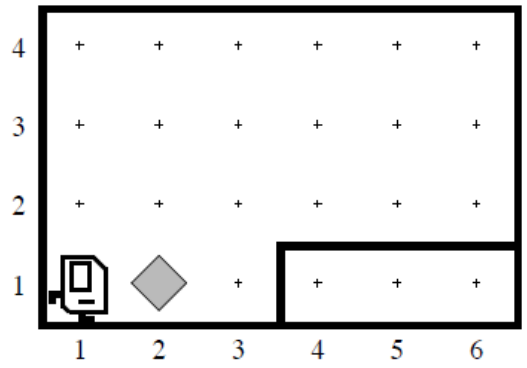


Control Statements in Karel

- Scenario 1: deterministic repetition
- Scenario 2: nondeterministic repetition
- Scenario 3: if else statement

Scenario 1: Deterministic Repetition

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

    public void turnRight()
        turnLeft();
        turnLeft();
        turnLeft();
    }

}
```



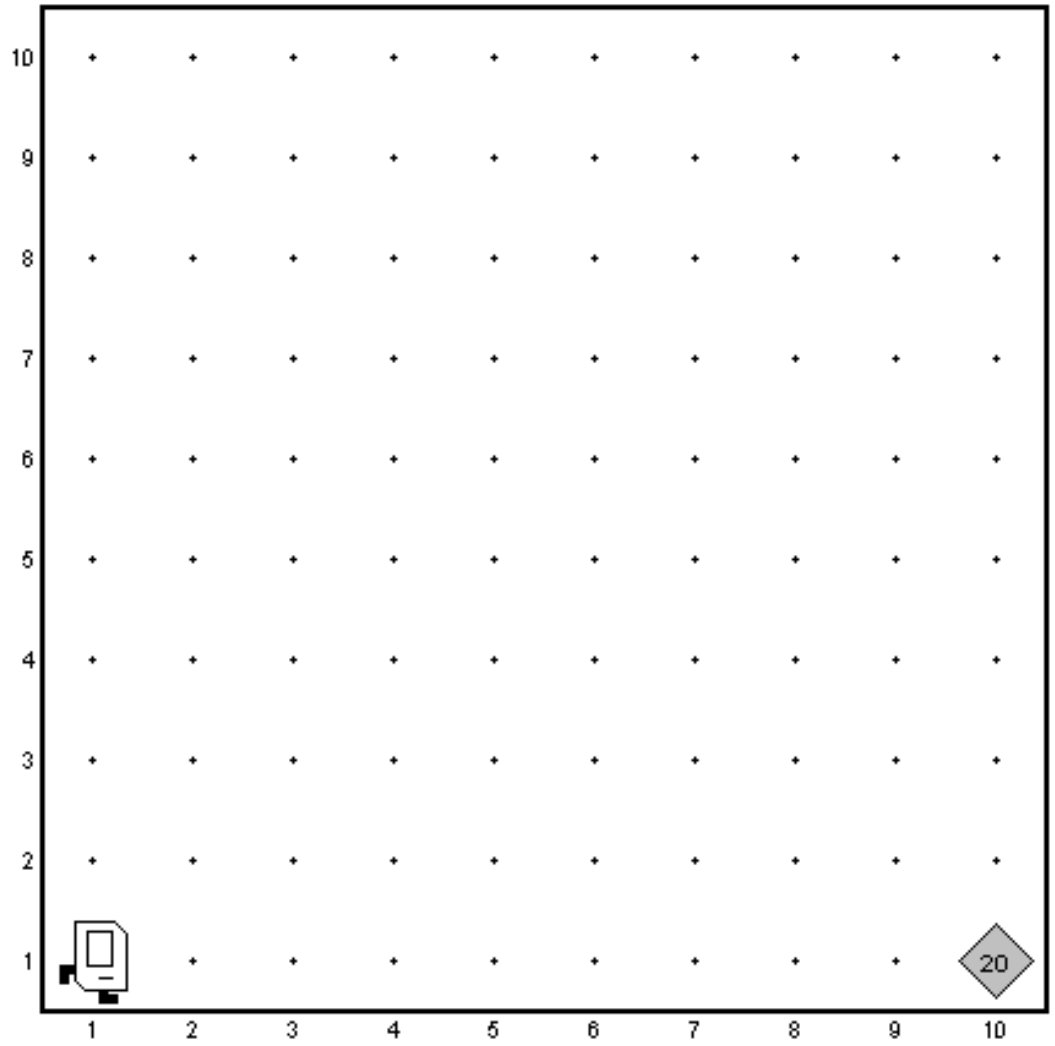
Karel is doing something a certain number of times!

=

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

Remember this syntax, you are going to use this all the time in Java

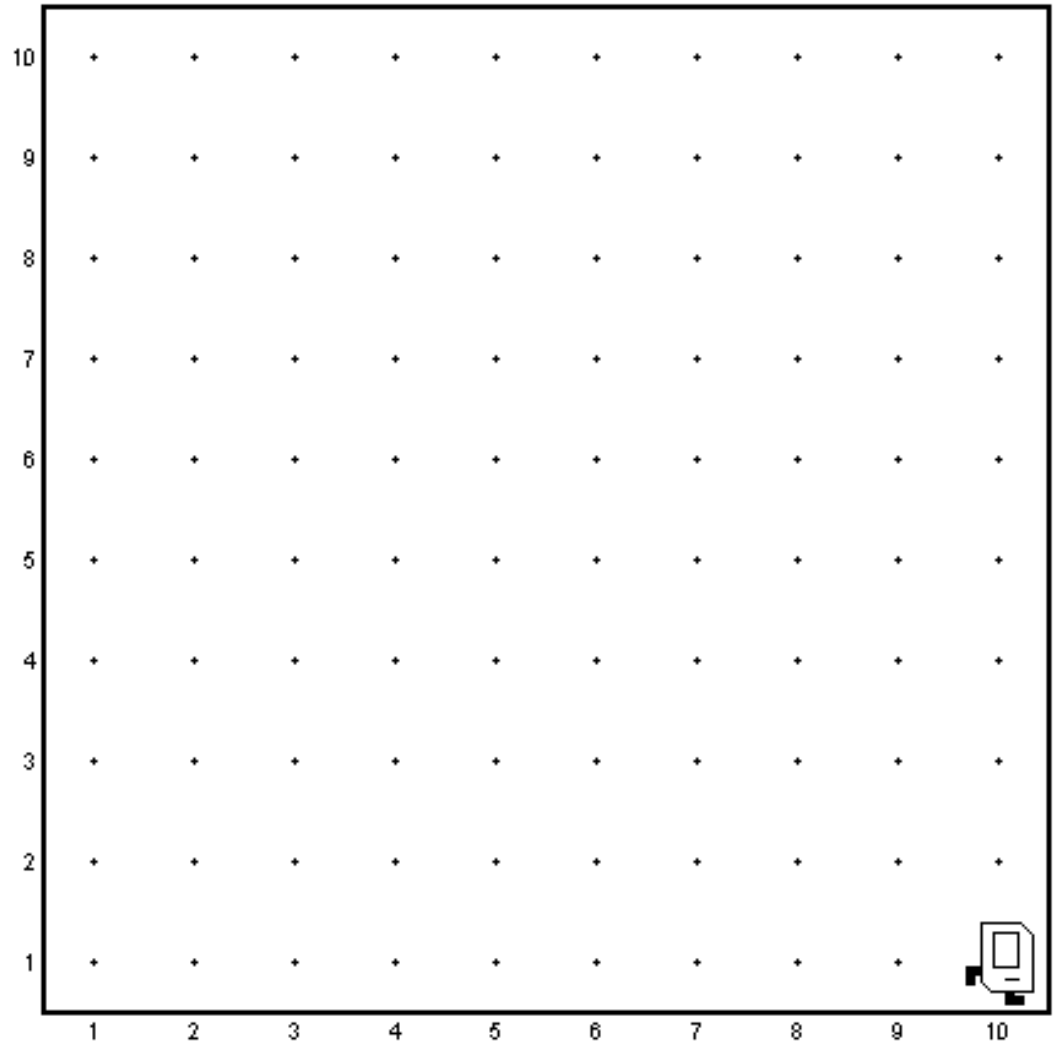
Practice 3



Move Karel to (10, 1) and let it pick up all the beepers on that corner!!! Use “for” loop.

Using the world “LoopyKarelWorld”;

Practice 4

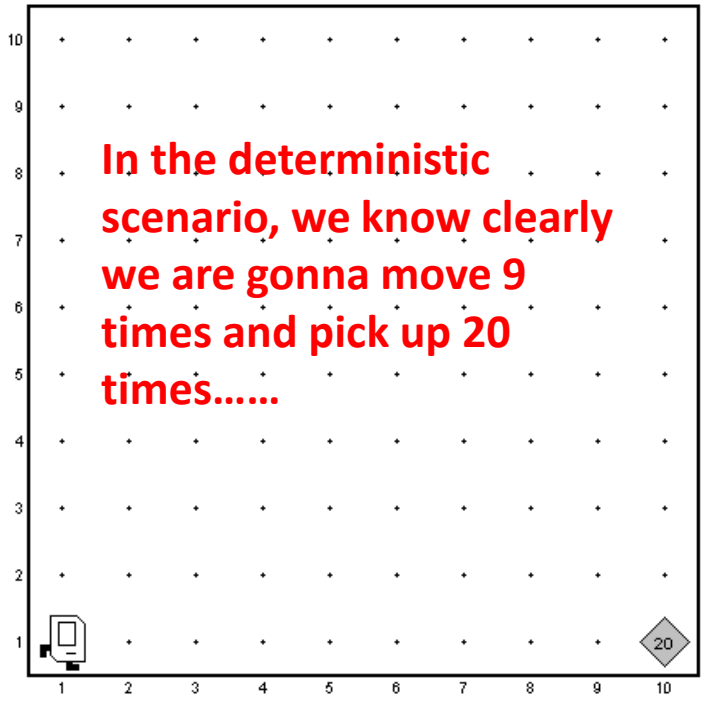


After following the instructions in practice 3, move Karel to (1, 10) and let it put 10 beepers on that corner!!! Use “for” loop.

Using the world “LoopyKarelWorld”;

Scenario 2: Nondeterministic Repetition

- What if there is some situation where you don't know how many times beforehand you want to do something.



In this case, Karel becomes very very cautious about the unknown world, like an ant, he has a feeler by which he can sense the world

Do Something While Some Condition Is True

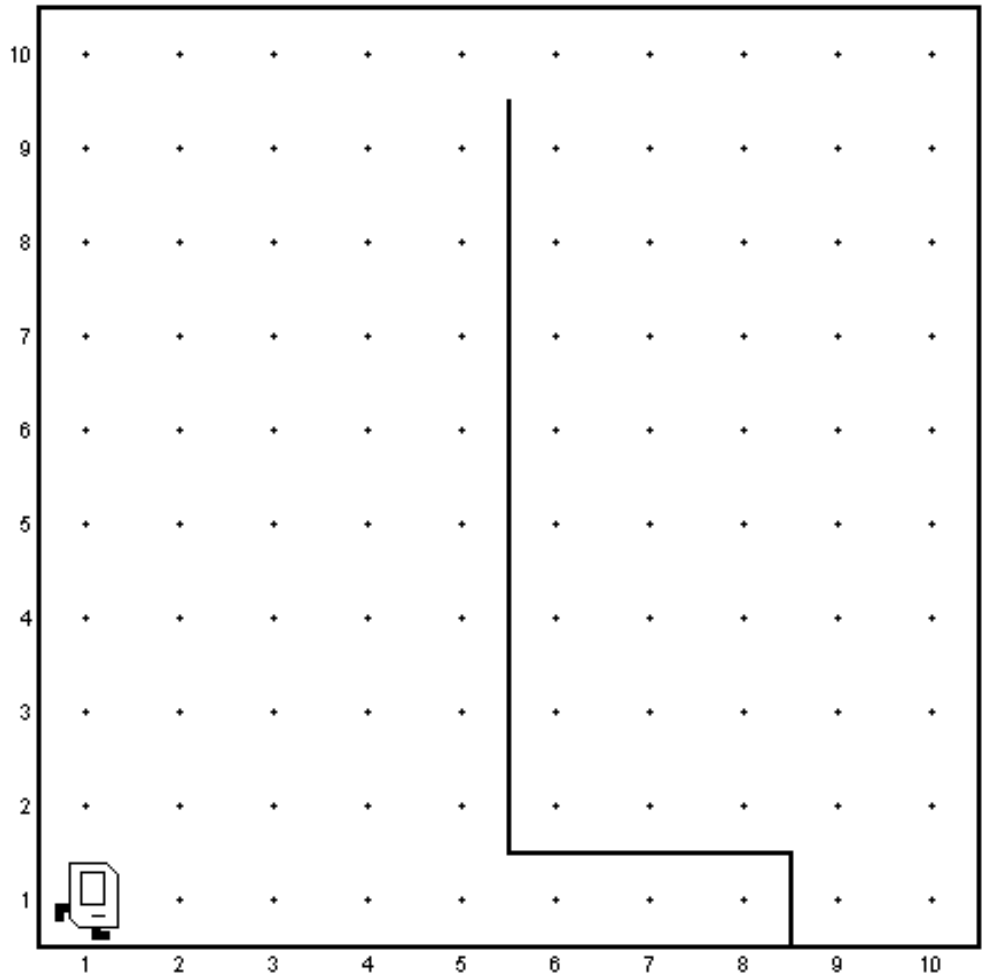
Keep going forward, as long as there is no wall in your way, when there is, stop.

The way we do that is something called a while loop:

```
while(condition){
    move();
}
```

Where does this come from?

Do Something While Some Condition Is True

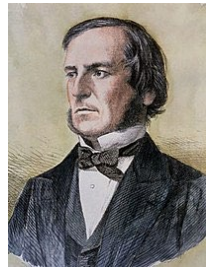



```
import stanford.karel.SuperKarel;
public class LoopyKarel extends SuperKarel{

    public void run(){
        moveToWall();
    }

    public void moveToWall(){
        while(frontIsClear()){
            move();
        }
    }

}
```

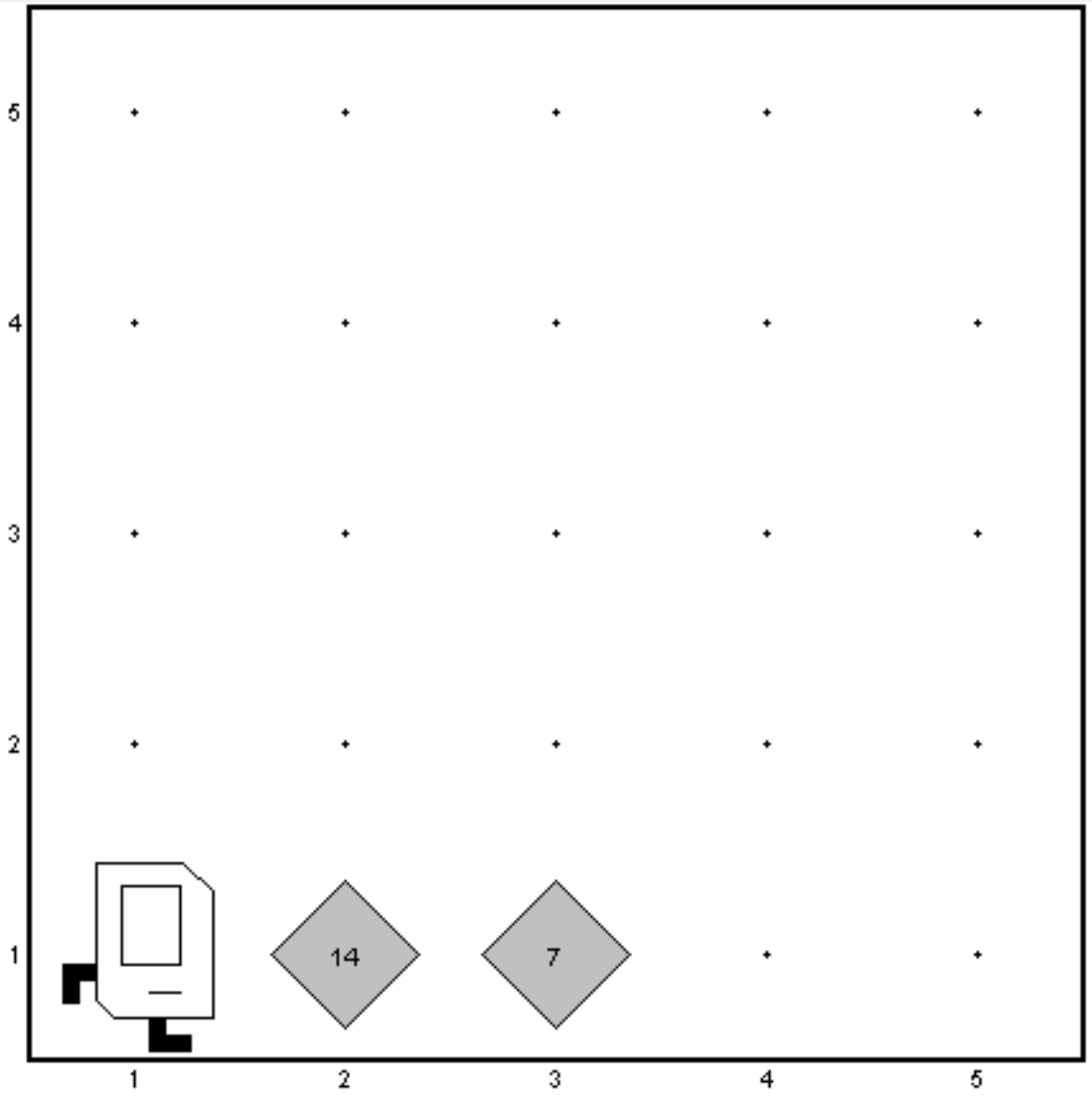
Which Are The Conditions Karel Can Check in His World?

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?

These are all method calls, they will return true or false.
True and false are called Boolean values after George Boole.

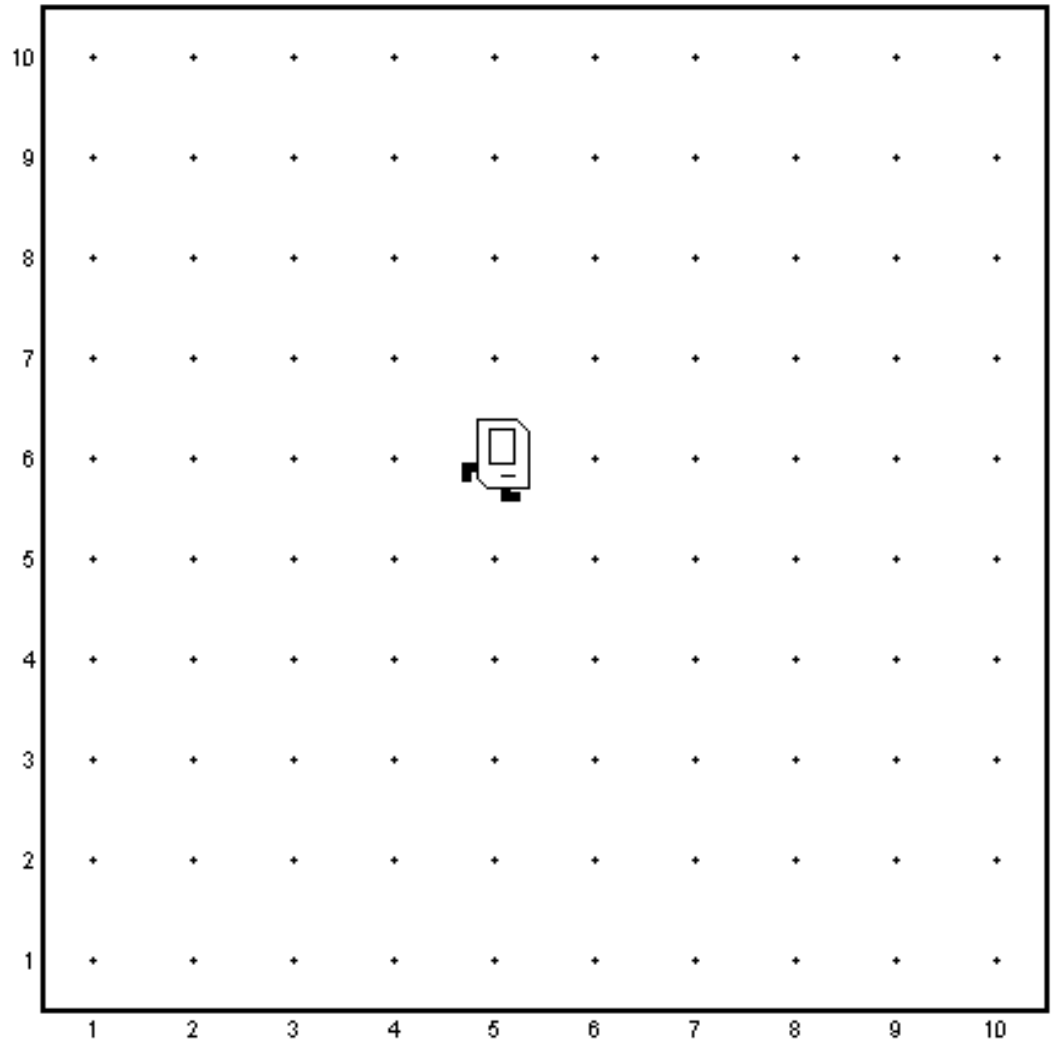


Practice 5



Using while loop, let Karel pick up the two piles of beepers

Infinite Loop



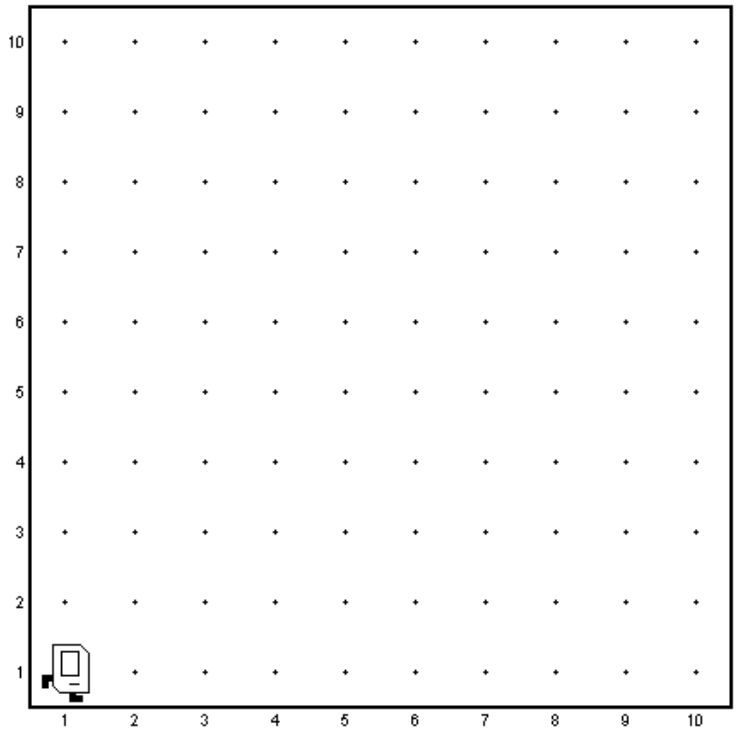

```
while(leftIsClear()){
    turnLeft();
}
```

When you are writing loops, make sure the loop will finally stop!!!

Check it after you are done with the loop every time.

Scenario 3: if else Statement

- Move to the wall, then stop, check:
  - if there is any beeper on that corner, pick one;
  - Else put one on that corner;



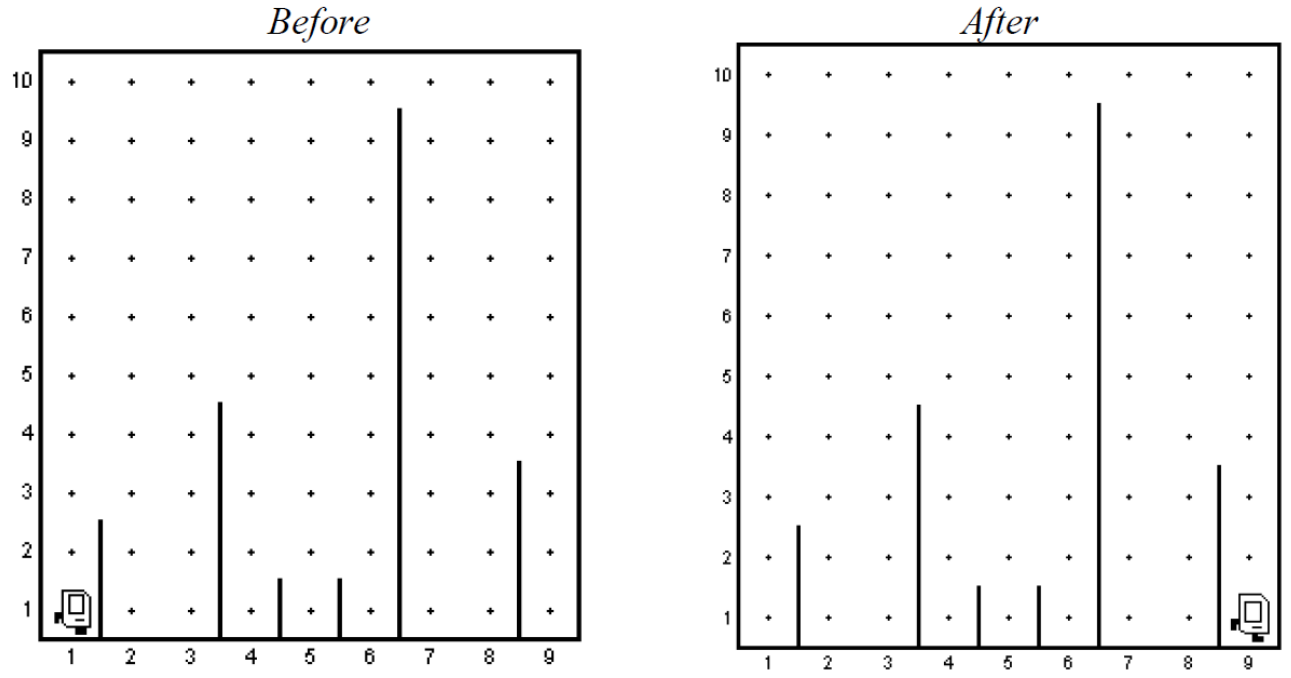

```
import stanford.karel.SuperKarel;
public class IfElseKarel extends SuperKarel{
    public void run(){
        moveToWall();//Here the control flow will jump to
                     //the method definition part
        if(beepersPresent()){
            pickBeeper();
        }
        else {
            putBeeper();
        }
    }
    public void moveToWall(){
        while(frontIsClear()){
            move();
        }
    }//after execution of this method, flow jumps back to
     //where the method gets called in run()
     //and pick up the statement after the method call
}
```

In this world on the left, beepersPresent() method call returns false!!!

Karel On Tough Problems

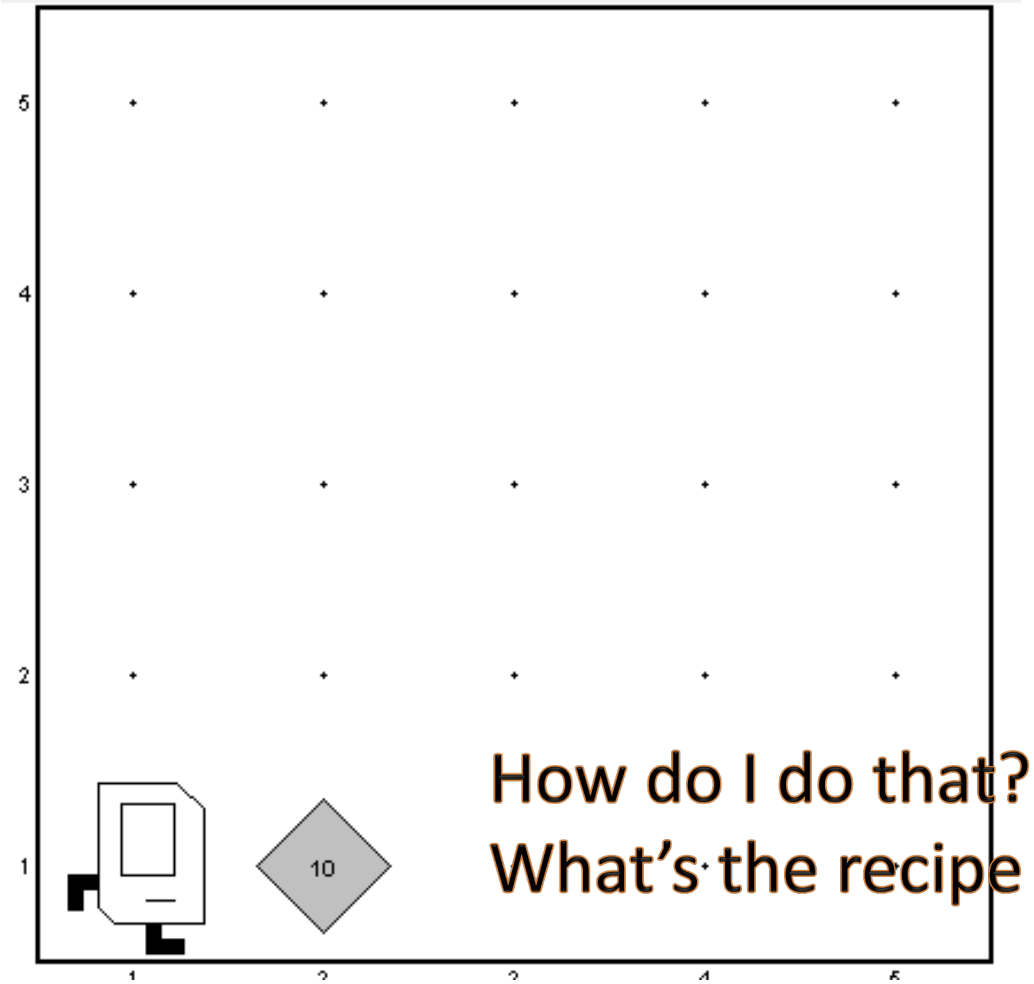
- Running a steeple chase
- Doubling the number of beepers in a pile

A Slightly Complicated Problem 1
Running A Steeple chase



- 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.

A Slightly Complicated Problem 2
Double Beepers

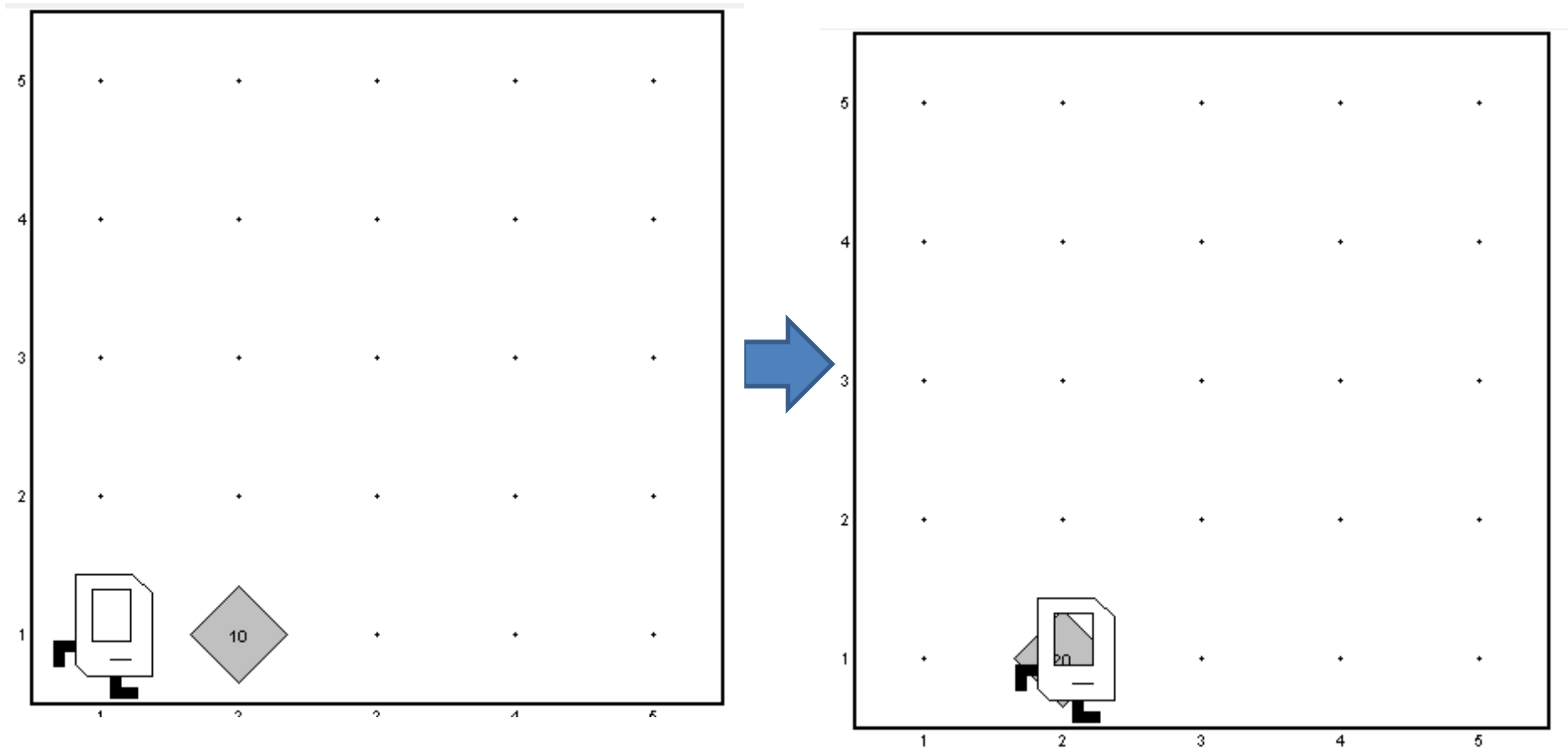


Write a program to let Karel double the number of beepers in front of him.

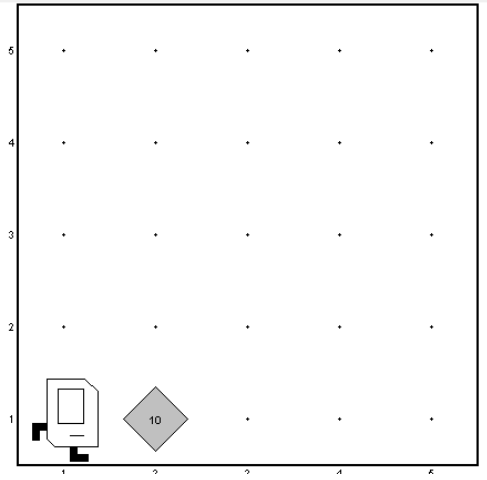
ATTENTION: Karel doesn't have a counter. But although he cannot count, he can still accomplish this task, Karel is great!

How do I do that?
What's the recipe for doing that?

A Slightly Complicated Problem- Double Beepers(cont'd)



Top-down Design



- move();
- doubleTheBeeper();
  - putDoubleBeepersOnNextCorner();
    - pickBeeper();
    - putTwoBeepersOnNextCorner();
      - move();
      - Invoke putBeeper() twice;
      - moveBackOneCorner();
        - » turnAround();
        - » move();
        - » turnAround();
  - moveBeepersOnNextCornerBack();
    - move()
    - pickBeeper();
    - moveBackOneCorner();
    - putBeeper();
    - moveBackOneCorner();

As long as there is beeper left

As long as there is beeper left

Primitives:
move();
turnLeft();
turnAround();
pickBeeper();
putBeeper();

doubleTheBeeper();

putDoubleBeepersOnNextCorner();

moveBeepersOnNextCornerBack();

Why we take things out to form a new method?

Because the things we take out are logically related!!!

```
public void run() {
    // the first move
    move();
    // putDoubleBeepersOnNextCorner()
    while (beepersPresent()) {
        pickBeeper();

        move();
        putBeeper();
        putBeeper();

        // moveBackOneCorner()
        turnAround();
        move();
        turnAround();
    }
    move();// move to the pile that has twice as many beepers
    // moveBeepersOnNextCornerBack()
    while (beepersPresent()) {
        pickBeeper();

        // moveBackOneCorner()
        turnAround();
        move();
        turnAround();

        putBeeper();
        move();
    }
    // moveBackOneCorner()
    turnAround();
    move();
    turnAround();
}
```

A Couple Things Related to That

- Each method solves one problem.
- Good method names should describe what the method is actually doing.
- Comments associated with method, make it self-explanatory.