4. The PR2004 is a robot that automatically gathers toys and other items scattered in a tiled hallway. A tiled hallway has a wall at each end and consists of a single row of tiles, each with some number of items to be gathered.

   The PR2004 robot is initialized with a starting position and an array that contains the number of items on each tile. Initially the robot is facing right, meaning that it is facing toward higher-numbered tiles.

   The PR2004 robot makes a sequence of moves until there are no items remaining on any tile. A move is defined as follows.
   1. If there are any items on the current tile, then one item is removed.
   2. If there are more items on the current tile, then the robot remains on the current tile facing the same direction.
   3. If there are no more items on the current tile
      a) if the robot can move forward, it advances to the next tile in the direction that it is facing;
      b) otherwise, if the robot cannot move forward, it reverses direction and does not change position.

   In the following example, the position and direction of the robot are indicated by "<" or ">" and the entries in the diagram indicate the number of items to be gathered on each tile. There are four tiles in this hallway. The starting state of the robot is illustrated in the following diagram.

Tile number:		0	1	2	3	
Number of items:	left wall →	**1**	**1**	**2**	**2**	← right wall
Robot position:			>			

   The following sequence shows the configuration of the hallway and the robot after each move.

After move 1

0	1	2	3
1	**0**	**2**	**2**
		>	

After move 2

0	1	2	3
1	**0**	**1**	**2**
		>	

After move 3

0	1	2	3
1	**0**	**0**	**2**
			>

After move 4

0	1	2	3
1	**0**	**0**	**1**
			>

After move 5

0	1	2	3
1	**0**	**0**	**0**
			<

After move 6

0	1	2	3
1	**0**	**0**	**0**
		<	

After move 7

0	1	2	3
1	**0**	**0**	**0**
	<		

After move 8

0	1	2	3
1	**0**	**0**	**0**
<			

After move 9

0	1	2	3
0	**0**	**0**	**0**
>			

   After nine moves, the robot stops because the hall is clear.

The PR2004 is modeled by the class `Robot` as shown in the following declaration.

```
public class Robot
{
  private int[] hall;
  private int pos;                // current position(tile number) of Robot
  private boolean facingRight; // true means this Robot is facing right


  // constructor not shown


  // postcondition: returns true if this Robot has a wall immediately in
  //                front of it, so that it cannot move forward;
  //                otherwise, returns false
  private boolean forwardMoveBlocked()
  {  /* to be implemented in part (a) */  }


  // postcondition: one move has been made according to the
  //                specifications above and the state of this
  //                Robot has been updated
  private void move()
  {  /* to be implemented in part (b) */  }


  // postcondition: no more items remain in the hallway;
  //                returns the number of moves made
  public int clearHall()
  {  /* to be implemented in part (c) */  }


  // postcondition: returns true if the hallway contains no items;
  //                otherwise, returns false
  private boolean hallIsClear()
  {  /* implementation not shown */  }
}
```

In the `Robot` class, the number of items on each tile in the hall is stored in the corresponding entry in the array `hall`. The current position is stored in the instance variable `pos`. The `boolean` instance variable `facingRight` is `true` if the `Robot` is facing to the right and is `false` otherwise.

(a) Write the `Robot` method `forwardMoveBlocked`. Method `forwardMoveBlocked` returns `true` if the robot has a wall immediately in front of it, so that it cannot move forward. Otherwise, `forwardMoveBlocked` returns `false`.

Complete method `forwardMoveBlocked` below.

```
// postcondition: returns true if this Robot has a wall immediately in
//                front of it, so that it cannot move forward;
//                otherwise, returns false
private boolean forwardMoveBlocked()
```

GO ON TO THE NEXT PAGE.

Question 4

Part A: `forwardMoveBlocked` 1 pt

- **+1** return boolean
  - **+1/2** check a dir/pos pair
  - **+1/2** correct

Part B: `move` 5 pts

- **+1** check for item(s) on current tile and remove one
  - **+1/2** attempt on current tile (might try to remove all items)
  - **+1/2** correct

- **+1 1/2** check required conditions in context of attempt to move/turn (body of each check must refer to `pos` or `facingRight`)
  - **+1** separate check for empty tile (e.g., not in ELSE)
  - **+1/2** check `forwardMoveBlocked`

- **+1** change direction (set direction to some value relative to current direction)
  - **+1/2** toggle value
  - **+1/2** if and only if originally blocked

- **+1 1/2** move (set position to value(s) relative to current position)
  - **+1/2** attempt 2 directions (change position, not value at position)
  - **+1/2** move 1 tile in proper direction
  - **+1/2** if and only if originally not blocked

Part C: `clearHall` 3 pts

- **+1/2** declare and initialize counter (must have some extra context relevant to counting)

- **+1** loop until done
  - **+1/2** call to `hallIsClear` in loop
  - **+1/2** correct

- **+1** robot action (in context of a loop)
  - **+1/2** call `move`
  - **+1/2** correctly determine number of times `move` is called

- **+1/2** always return number of times `move` is called (no credit for returning 0 with no call to `move` in code)

2004 General Usage/Java

Most common usage errors are addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once on a part when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem, even if it occurs on different parts of a problem.

Non-penalized Errors

case discrepancies

variable not declared when others are declared in some part of question

missing "new" for constructor call once, when others are present in question

default constructor called without parens for example, `new Fish;`

missing { } where indentation clearly conveys intent

`obj.method` instead of `obj.method()`

loop variables used outside loop

`[r,c]`, `(r)(c)` or `(r,c)` instead of `[r][c]`

= instead of `==` (and vice versa)

missing `( )` around `if/while` conditions

`length` - `size` confusion for array, `String`, and `ArrayList`, with or without `()`

missing downcast from collection or map

unnecessary construction of object whose reference is reassigned, for example

```
Direction dir = new Direction();
dir = f.Direction;
```

`private` qualifier on local variable

use "," instead of "+" for `String` in `System.out.print(str1, str2)`)

missing `;`s or missing `public`

extraneous code with no side-effect, for example a check for precondition

automatic conversion of `Integer` to `int` and vice-versa (this is legal in Java 1.5, called auto(un)boxing)

Minor Errors (1/2 point)

misspelled/ confused identifier (e.g., `len` for `length` or `left()` for `getLeft()`)

no variables declared

`new` never used for constructor calls

`void` method returns a value

modifying a constant (`final`)

use `equals` or `compareTo` method on primitives, for example `int x; …x.equals(val)`

use value 0 for `null`

use values 0, 1 for `false`, `true`

use of `itr.next()` more than once as same value within loop

use keyword as identifier

`[]` – `get` confusion

assignment dyslexia, for example,
`x + 3 = y;` for `y = x + 3;`

Major Errors (1 point)

read new values for parameters or or instance variables (prompts part of this point)

extraneous code which causes side-effect, for example, information written to output.

use interface or class name instead of variable identifier, for example `Simulation.step()` instead of `sim.step()`

`aMethod(obj)` instead of `obj.aMethod()`

use of object reference that is incorrect, for example, use of `f.move()` inside method of `Fish` class

use private data or method when not accessible

destruction of data structure (e.g. by using root reference to a `TreeNode` for traversal of the tree; this is often handled in the rubric)

Note: Case discrepancies for identifiers fall under the "not penalized" category. However, if they result in another error, they must be penalized. Sometimes students bring this on themselves with their definition of variables. For example, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the one point deduction applies. Interpret writing to give benefit of the doubt to the student.