3. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

Consider adding a `PondStocker` class to the Marine Biology Simulation case study. The purpose of the `PondStocker` class is to increase the fish population in a pond if the population density falls below a certain minimum. The population density of an environment is the ratio of the number of fish in the environment to the total number of locations in the environment. When there are too few fish in the environment, enough fish will be added to make the population greater than the specified minimum density. You will be asked to implement some of the methods for the `PondStocker` class. The declaration of the `PondStocker` class is as follows.

```
public class PondStocker
{
  private Environment theEnv;
  private double minDensity;    // 0.0 <= minDensity < 1.0

  // postcondition: returns the minimum number of fish that need to be
  //                added to make the population density greater than
  //                minDensity
  private int numUnder()
  {  /* to be implemented in part (a) */  }

  // postcondition: returns a random location within the bounds of theEnv
  private Location randomLocation()
  {  /* to be implemented in part (b) */  }

  // precondition:  0 <= numToAdd <= number of empty locations in theEnv
  // postcondition: the number of fish in theEnv has been increased
  //                by numToAdd; the fish added are placed at
  //                random empty locations in theEnv
  public void addFish(int numToAdd)
  {  /* to be implemented in part (c) */  }

// constructor and other methods not shown
}
```

For example, suppose that the environment has 7 rows and 7 columns, giving it a total of 49 cells. If the minimum density is 0.5, 25 cells need to be occupied to meet the minimum density requirement. If the number of fish in the environment is 17, then the call `numUnder()` would return 8.

GO ON TO THE NEXT PAGE.

(a) Write the `PondStocker` method `numUnder`. Method `numUnder` returns the smallest number of fish that must be added to make the density of fish in the environment greater than `minDensity`. If the density of fish in the environment is already greater than `minDensity`, then `numUnder` returns zero. Recall that the `Environment` methods `numRows` and `numCols` return the number of rows and the number of columns, respectively, in an environment.

Complete method `numUnder` below.

```
// postcondition: returns the minimum number of fish that need to be
//                added to make the population density greater than
//                minDensity
private int numUnder()
```

(b) Write the `PondStocker` method `randomLocation`. Method `randomLocation` returns a random location within the bounds of the environment.

In writing `randomLocation`, you may use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `randomLocation` below.

```
// postcondition: returns a random location within the bounds of theEnv
private Location randomLocation()
```

(c) Write the `PondStocker` method `addFish`. Method `addFish` adds `numToAdd Fish` to the environment at random locations that are not already occupied. You may use the two-parameter `Fish` constructor, so that the fish added have a random direction and color.

In writing `addFish`, you may call `randomLocation`. Assume that `randomLocation` works as specified, regardless of what you wrote in part (b). You may also use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `addFish` below.

```
// precondition:  0 <= numToAdd <= number of empty locations in theEnv
// postcondition: the number of fish in theEnv has been increased
//                by numToAdd; the fish added are placed at
//                random empty locations in theEnv
public void addFish(int numToAdd)
```

GO ON TO THE NEXT PAGE.

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.

GO ON TO THE NEXT PAGE.