1. The following class `WordList` is designed to store and manipulate a list of words. The incomplete class declaration is shown below. You will be asked to implement two methods.

```
public class WordList
{
  private ArrayList myList;  // contains Strings made up of letters


  // postcondition: returns the number of words in this WordList that
  //                are exactly len letters long
  public int numWordsOfLength(int len)
  {  /* to be implemented in part (a) */  }


  // postcondition: all words that are exactly len letters long
  //                have been removed from this WordList, with the
  //                order of the remaining words unchanged
  public void removeWordsOfLength(int len)
  {  /* to be implemented in part (b) */  }


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

(a) Write the `WordList` method `numWordsOfLength`. Method `numWordsOfLength` returns the number of words in the `WordList` that are exactly `len` letters long. For example, assume that the instance variable `myList` of the `WordList animals` contains the following.

```
["cat", "mouse", "frog", "dog", "dog"]
```

The table below shows several sample calls to `numWordsOfLength`.

| Call | Result returned by call |
|---|---|
| `animals.numWordsOfLength(4)` | 1 |
| `animals.numWordsOfLength(3)` | 3 |
| `animals.numWordsOfLength(2)` | 0 |

Complete method `numWordsOfLength` below.

```
  // postcondition: returns the number of words in this WordList that
  //                are exactly len letters long
  public int numWordsOfLength(int len)
```

(b) Write the `WordList` method `removeWordsOfLength`. Method `removeWordsOfLength` removes all words from the `WordList` that are exactly `len` letters long, leaving the order of the remaining words unchanged. For example, assume that the instance variable `myList` of the `WordList` `animals` contains the following.

```
["cat", "mouse", "frog", "dog", "dog"]
```

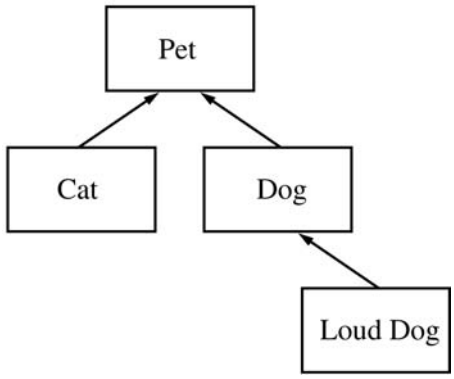
The table below shows a sequence of calls to the `removeWordsOfLength` method.

| Call | `myList` after the call |
|---|---|
| `animals.removeWordsOfLength(4);` | `["cat", "mouse", "dog", "dog"]` |
| `animals.removeWordsOfLength(3);` | `["mouse"]` |
| `animals.removeWordsOfLength(2);` | `["mouse"]` |

Complete method `removeWordsOfLength` below.

```
// postcondition: all words that are exactly len letters long
//                have been removed from this WordList, with the
//                order of the remaining words unchanged
public void removeWordsOfLength(int len)
```

2. Consider the hierarchy of classes shown in the following diagram.




Note that a `Cat` "is-a" `Pet`, a `Dog` "is-a" `Pet`, and a `LoudDog` "is-a" `Dog`.

The class `Pet` is specified as an abstract class as shown in the following declaration. Each `Pet` has a name that is specified when it is constructed.

```
public abstract class Pet
{
  private String myName;

  public Pet(String name)
  {  myName = name;  }

  public String getName()
  {  return myName;  }

  public abstract String speak();
}
```

The subclass `Dog` has the partial class declaration shown below.

```
public class Dog extends Pet
{
  public Dog(String name)
  {  /* implementation not shown */  }

  public String speak()
  {  /* implementation not shown */  }
}
```

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()
```

(b) Write the `Robot` method `move`. Method `move` has the robot carry out one move as specified at the beginning of the question. The specification for a move is repeated here for your convenience.

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 writing `move`, you may use any of the other methods in the `Robot` class. Assume these methods work as specified, regardless of what you wrote in part (a). Solutions that reimplement the functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `move` below.

```
// postcondition: one move has been made according to the
//                specifications above and the state of this
//                Robot has been updated
private void move()
```

(c) Write the `Robot` method `clearHall`. Method `clearHall` clears the hallway, repeatedly having this robot make a move until the hallway has no items, and returns the number of moves made.

In the example at the beginning of this problem, `clearHall` would take the robot through the moves shown and return 9, leaving the robot in the state shown in the final diagram.

In writing `clearHall`, you may use any of the other methods in the `Robot` class. Assume these methods work as specified, regardless of what you wrote in parts (a) and (b). Solutions that reimplement the functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `clearHall` below.

```
// postcondition: no more items remain in the hallway;
//                returns the number of moves made
public int clearHall()
```

**END OF EXAMINATION**

# AP® Computer Science A
# 2004 SCORING GUIDELINES

## Question 1

**Part A:** `numWordsOfLength` **4 pts**

**+1/2** declare and initialize count to zero (could be an empty list, length = 0)
(must show evidence that variable is used for counting or returned)

**+1** loop over `myList`
- **+1/2** attempt (must reference `myList` in body)
- **+1/2** correct

**+1/2** get `String` from `myList` (no deduction for missing downcast but local must be `String`)
(lose this if array syntax used)

**+1** check length of `String`
- **+1/2** attempt (must be in context of loop)
- **+1/2** correct (array syntax is OK)

**+1/2** increment count (must be within context of length check)
(lose this if count does not accumulate)

**+1/2** return correct count (after loop is completed)

**Part B:** `removeWordsOfLength` **5 pts**

**+2** loop over `myList`
- **+1** attempt (must reference `myList` in body)
- **+1** correct (must have attempt at removal, must not skip items)

**+1** get `String` from `myList` (no deduction for missing downcast, but local must be `String`)
- **+1/2** attempt (must be in context of loop, array syntax is OK)
- **+1/2** correct (no array syntax)

**+1** check length of `String`
- **+1/2** attempt (must be in context of loop)
- **+1/2** correct (array syntax is OK)

**+1** remove
- **+1/2** attempt (must call `remove`, must refer to `myList` or an index of an element in `myList`)
- **+1/2** correct (no array syntax)

**Usage**:

**-1/2** for `WordList` instead of `myList`

**-1/2** for returning a value in part B

**-1** for using `this` instead of `myList`, can lose in part A and again in part B (for max of -2)