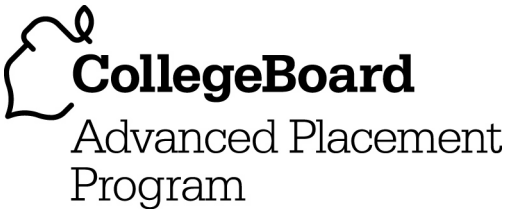



# AP® Computer Science A
# 2004 Free-Response Questions

**COMPUTER SCIENCE A**
**SECTION II**
**Time—1 hour and 45 minutes**
**Number of questions—4**
**Percent of total grade—50**

**Directions: SHOW ALL YOUR WORK, REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN Java.**

Notes:
- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.

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"]
```

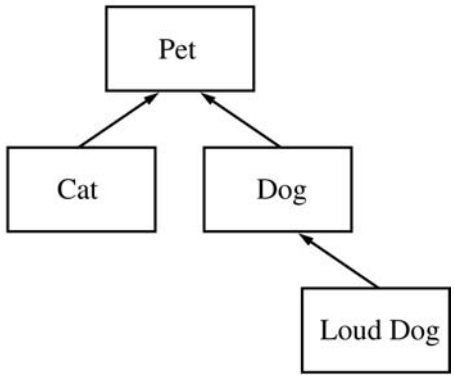
The table below shows a sequence of calls to the `removeWordsOfLength` method.

| 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 */  }
}
```

(a) Given the class hierarchy shown above, write a complete class declaration for the class `Cat`, including implementations of its constructor and method(s). The `Cat` method `speak` returns `"meow"` when it is invoked.

(b) Assume that class `Dog` has been declared as shown at the beginning of the question. If the `String` *dog-sound* is returned by the `Dog` method `speak`, then the `LoudDog` method `speak` returns a `String` containing *dog-sound* repeated two times.

Given the class hierarchy shown previously, write a complete class declaration for the class `LoudDog`, including implementations of its constructor and method(s).

(c) Consider the following partial declaration of class `Kennel`.

```
public class Kennel
{
  private ArrayList petList;     // all elements are references
                                 // to Pet objects

  // postcondition: for each Pet in the kennel, its name followed
  //                by the result of a call to its speak method
  //                has been printed, one line per Pet
  public void allSpeak()
  {  /* to be implemented in this part */  }

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

Write the `Kennel` method `allSpeak`. For each `Pet` in the kennel, `allSpeak` prints a line with the name of the `Pet` followed by the result of a call to its `speak` method.

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

Complete method `allSpeak` below.

```
// postcondition: for each Pet in the kennel, its name followed
//                by the result of a call to its speak method
//                has been printed, one line per Pet
public void allSpeak()
```

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.

(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.

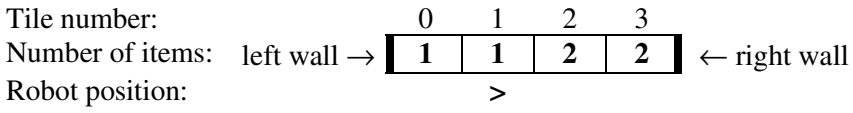
```
// precondition:  0 <= numToAdd <= number of empty locations in theEnv
// postcondition: the number of fish in theEnv has been increased
//                by numToAdd; the fish added are placed at
//                random empty locations in theEnv
public void addFish(int numToAdd)
```

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.

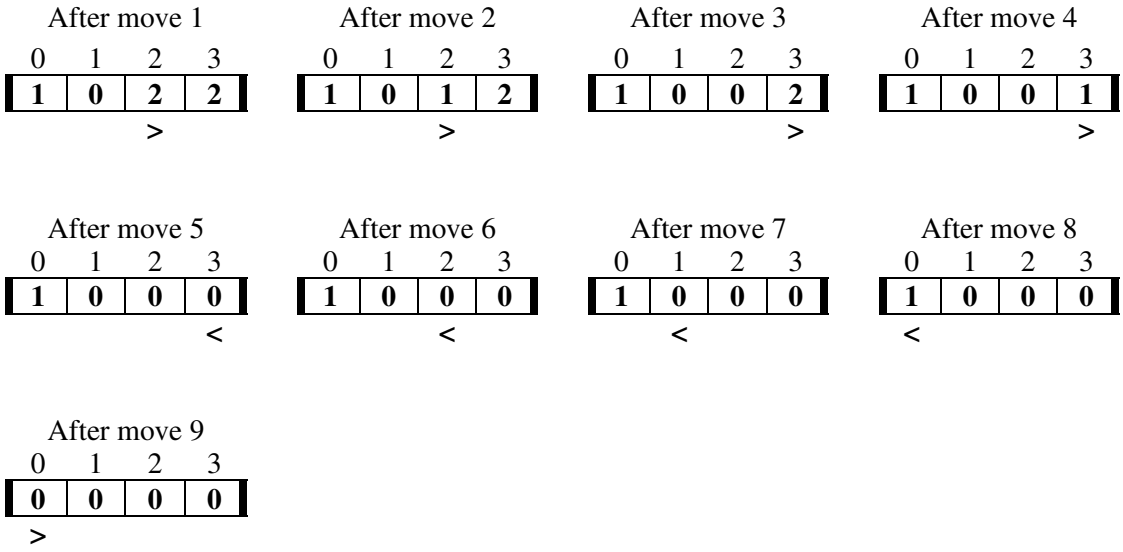
   The PR2004 robot is initialized with a starting position and an array that contains the number of items on each tile. Initially the robot is facing right, meaning that it is facing toward higher-numbered tiles.

   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.




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




   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()
```

(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**