Question 4: Path Finder (MBS)

Part A:	`possibleEnds`	**2 1/2 points**

- **+1** check ends to see if empty
  - **+1/2** attempt to check if one end is empty
  - **+1/2** correctly check both ends using `isEmpty` or `objectAt`
- **+1** check ends for direction
  - **+1/2** attempt to compare # rows or cols of `start & end`
  - **+1/2** correctly compare # rows & cols
- **+1/2** return correct value

Part B:	`findNEPath`	**6 1/2 points**

- **+1** check endpoints
  - **+1/2** call `possibleEnds(start, end)`
  - **+1/2** return `null` if fails
- **+2** base case (when `start` equals `end`)
  - **+1/2** compare to see if `start.equals(end)` (both must be empty)
  - **+1/2** create `List`
  - **+1/2** add `start/end` to list
  - **+1/2** return list
- **+1 1/2** recursive cases
  - **+1/2** attempt recursive call from neighbor of `start` (N or E) or `end` (S or W)
  - **+1/2** attempt other recursive call whenever first case fails
  - **+1/2** correct calls for both directions
- **+2** recursive application
  - **+1** add `start/end` to recursive path
    - **+1/2** attempt (must have `start/end` and recursive path)
    - **+1/2** correct
  - **+1/2** return updated path
  - **+1/2** return `null` if and only if no path exists

AP® COMPUTER SCIENCE A/AB
2006 GENERAL USAGE

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

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

A usage error that occurs once 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.

Nonpenalized Errors

spelling/case discrepancies*

local variable not declared when any other variables are declared in some part

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

use keyword as identifier

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

= instead of == (and vice versa)

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

`private` qualifier on local variable

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

common mathematical symbols for operators (x • ÷ ≤ ≥ <> ≠)

missing { } where indentation clearly conveys intent

missing `( )` on method call or around `if/while` conditions

missing `;`s

missing “new” for constructor call once, when others are present in some part

missing downcast from collection

missing `int` cast when needed

missing `public` on class or constructor header

Minor Errors (1/2 point)

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

no local variables declared

`new` never used for constructor calls

`void` method or constructor returns a value

modifying a constant (`final`)

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

`[] – get` confusion if access not tested in rubric

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

`super(method())` instead of `super.method()`

formal parameter syntax (with type) in method call, e.g., `a = method(int x)`

missing `public` from method header when required

"false"/"true" or 0/1 for boolean values

"null" for `null`

Major Errors (1 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)

use class name in place of `super` either in constructor or in method call

**Note: Spelling and case discrepancies for identifiers fall under the "nonpenalized" category as long as the correction can be unambiguously inferred from context. For example, "Queu" instead of "Queue". Likewise, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the context allows for the reader to assume the object instead of the class.*

Question 4: Path Finder (MBS)

PART A:

```
private boolean possibleEnds(Location start, Location end)
{
  return (theEnv.isEmpty(start) && theEnv.isEmpty(end) &&
          start.row() >= end.row() && start.col() <= end.col());
}
```

PART B:

Note: Commented code represents a more common approach but utilizes a `List` *method not in the APCS Quick Reference. Each commented line replaces the line above it in the alternate solution.*

```
public List findNEPath(Location start, Location end)
{
  if (!possibleEnds(start, end))
  {
    return null;
  }

  List path;
  if (start.equals(end))
  {
    path = new LinkedList();
    path.add(end);
    // path.add(0, start);
    return path;
  }

  path = findNEPath(start, theEnv.getNeighbor(end, Direction.SOUTH));
  // path = findNEPath(theEnv.getNeighbor(start, Direction.NORTH), end);


  if (path == null)
  {
    path = findNEPath(start, theEnv.getNeighbor(end, Direction.WEST));
    // path = findNEPath(theEnv.getNeighbor(start, Direction.EAST), end);
  }

  if (path != null)
  {
    path.add(end);
    // path.add(0, start);
  }

  return path;
}
```

AB4A,

(a) Write the PathFinder method possibleEnds, which returns true if the specified locations are possible end points for an NE-path. In order to be possible end points for an NE-path, both locations must be empty and the start location cannot be north or east of the end location.

Complete method possibleEnds below.

```
// returns true if the specified locations are possible
// starting and ending locations for an NE-path
// otherwise, returns false
private boolean possibleEnds(Location start, Location end) {
        if (! theEnv.isEmpty(start) || ! theEnv.isEmpty(end))
                return false;

        return   start.row()>=end.row()  &&  start.col()<= end.col();
}
```

AB4A2

(b) Write the `PathFinder` method `findNEPath`, which recursively searches the environment to find an NE-path between the locations `start` and `end`. If there is more than one NE-path connecting the locations, the method may return any one of those NE-paths. If there are no NE-paths connecting the locations, the method should return null.

In writing `findNEPath`, you may assume that `possibleEnds` works as specified regardless of what you wrote in part (a).

Complete method `findNEPath` below.

```
// returns a list containing a sequence of empty locations
// that form an NE-path from start to end;
// returns null if no such path exists
// postcondition: theEnv is unchanged
public List findNEPath(Location start, Location end) {
    if(!possibleEnds(start, end))
        return null;

    if(start.equals(end)) {
        List path = new LinkedList()
        path.add(start);
        return path;
    }
    List path = findNEPath(new Location(start.row()-1, start.col,
                end));

    if(path == null)
        path = findNEPath(new Location(start.row(),
                start.col()+1), end);

    if(path == null)
        return null;
    path.add(0, start);
    return path;

}
```

GO ON TO THE NEXT PAGE.

AB4B₁

(a) Write the `PathFinder` method `possibleEnds`, which returns `true` if the specified locations are possible end points for an NE-path. In order to be possible end points for an NE-path, both locations must be empty and the start location cannot be north or east of the end location.

Complete method `possibleEnds` below.

```
// returns true if the specified locations are possible
// starting and ending locations for an NE-path
// otherwise, returns false
private boolean possibleEnds(Location start, Location end)
{
  if( env.getDirection(start, end) != Direction.NORTHEAST ||
      env.getDirection(start, end) != Direction.NORTH ||
      env.getDirection(start, end) != Direction.EAST)
    return false;
  else if (env.isValid(start) && env.isEmpty(Start) &&
           env.isValid(end) && env.isEmpty(end))
      { return true;}
      else return false;

}
```

-20-

GO ON TO THE NEXT PAGE.

AB4B2

(b) Write the `PathFinder` method `findNEPath`, which recursively searches the environment to find an NE-path between the locations `start` and `end`. If there is more than one NE-path connecting the locations, the method may return any one of those NE-paths. If there are no NE-paths connecting the locations, the method should return null.

In writing `findNEPath`, you may assume that `possibleEnds` works as specified regardless of what you wrote in part (a).

Complete method `findNEPath` below.

```
// returns a list containing a sequence of empty locations
// that form an NE-path from start to end;
// returns null if no such path exists
// postcondition: theEnv is unchanged
public List findNEPath(Location start, Location end)
{ List NEpath = new ArrayList();
  if (!possibleEnds(start, end))
    return null;
  else
   { if (start.equals(end))
      {
       NEpath.add(end);
       return NEpath;
      }
     else { Location newNorthStart = new Location(start.row()+1, start.col());
         Location newEastStart = new Location(start.row(), start.col()+1
         if(possibleEnds(newNorthStart, end))
          {NEpath.add(findNEPath(newNorthStart, end));
           return NEPath;
          }
         else if(possibleEnds(NewEastStart, end)
            { NEpath.add(findNEPath(newEastStart, end));
              return NEpath;
            }
         }
   }
}
```

GO ON TO THE NEXT PAGE.

AB4C1

(a) Write the `PathFinder` method `possibleEnds`, which returns `true` if the specified locations are possible end points for an NE-path. In order to be possible end points for an NE-path, both locations must be empty and the start location cannot be north or east of the end location.

Complete method `possibleEnds` below.

```
// returns true if the specified locations are possible
// starting and ending locations for an NE-path
// otherwise, returns false
private boolean possibleEnds(Location start, Location end)
{ if ( theEnv.isEmpty(start) && theEnv.isEmpty(end))
  {   if((start.row()).compareTo(end.row()) > 0)
      {      if((start.col()).compareTo(end.col())<0)
             { return true }
         return false;
      }
    return false;
  }
  return false;
}
```

AB4C2

(b) Write the `PathFinder` method `findNEPath`, which recursively searches the environment to find an NE-path between the locations `start` and `end`. If there is more than one NE-path connecting the locations, the method may return any one of those NE-paths. If there are no NE-paths connecting the locations, the method should return null.

In writing `findNEPath`, you may assume that `possibleEnds` works as specified regardless of what you wrote in part (a).

Complete method `findNEPath` below.

```
// returns a list containing a sequence of empty locations
// that form an NE-path from start to end;
// returns null if no such path exists
// postcondition: theEnv is unchanged
public List findNEPath(Location start, Location end)
{ private List List = new List;
  list.add(Start.toString();)
  if (theEnv.isEmpty(row-1,col))
    list.add


  return List;
}
```

GO ON TO THE NEXT PAGE.

Question 4

Overview

This question was based on the Marine Biology Simulation case study and focused on abstraction, recursion, and algorithm implementation. Students needed to show their understanding of the case study and its interacting classes in order to recursively search for a path in an `Environment`. A skeleton of the `PathFinder` class was provided, which contained an `Environment` as a private field. A recursive algorithm for finding an NE-path between two `Locations` in the `Environment` was described, and students were required to implement that search algorithm. In part (a) students were required to complete the `private possibleEnds` method, which determined whether two `Locations` could be endpoints of an NE-path. This involved checking to see that both `Locations` were valid and that the ending `Location` was north and/or east of the starting `Location`. In part (b) students were required to complete the `findNEPath` method, which performed the recursive search between the two `Locations` and returned a path if one existed. This involved checking base cases (the two `Locations` were the same, or one was nonempty), performing recursive searches from neighboring locations, and collecting the path in a `List` of `Locations` when a path was found. While the algorithm for conducting the recursive search was specified, there was considerable freedom in the implementation. Determining whether two `Locations` were possible endpoints in part (a) was most commonly accomplished by comparing row and column numbers but also could be accomplished using the `getDirection` method from `Environment`. Likewise, the recursion in part (b) could be expanded forwards from the starting `Location` (using a north or east neighbor) or backwards from the ending `Location` (using a south or west neighbor).

Sample: AB4A
Score: 9

This solution correctly checks for both empty and direction and earned all the points available in part (a).

In part (b) the solution correctly uses `possibleEnds` and returns `null`, so both ½ points were earned. The solution correctly checks for `start` equal to `end`, creates a `List`, adds `start` and returns, which earned the next four ½ points for the base case. The solution creates a path to the north by correctly calling `findNEPath`. If this path is `null` the solution creates a path to the east. If `this` path is `null`, `null` is returned. If this path is not `null`, `start` is added at the beginning and the path is returned. This solution earned all of the remaining ½ points for a total score of 9.

Sample: AB4B
Score: 5

The solution attempts to check for direction but uses `||` instead of `&&` earning the ½ point for attempt but not the ½ point for correctness in the check ends for direction. The student attempts to check for empty but lost the correctness ½ point because of the confused identifier (`env` is used instead of `theEnv`). This student earned the ½ point for the return value.

The solution correctly uses and returns `possibleEnds`, which earned both of the check endpoints ½ points. The solution correctly creates a `List`, checks for equals, adds ends, and returns, earning all four ½ points for the base case. The solution attempts a recursive call from a neighbor but adds 1 to row so the path search is to the south, losing the ½ point for correctness. The guard on when to make the second call is on `possibleEnds` instead of failure to find a path, so the ½ point for call2 was not earned. The solution does not attempt to combine the result of

Question 4 (continued)

the recursive call with `start` and does not guarantee a return of `null` when no path exists. Therefore, all four ½ points in the recursive application portion of the scoring guidelines were lost.

Sample: AB4C
Score: 2

In part (a) the student correctly checks for empty and earned both ½ points for this check. The student attempts to check for direction but uses `compareTo`. `compareTo` works if checking against equality of `Locations`. If using `compareTo` with inequality, the result is incorrect because it is based on row major order. Thus, this solution earned the attempt ½ point but not the correctness ½ point. The solution earned the ½ point for return.

In part (b) the code earned no points as defined by the scoring guidelines.