AP® COMPUTER SCIENCE A
2006 SCORING GUIDELINES

Question 4: Drop Game (MBS)

Part A: `dropLocationForColumn` **3 1/2 points**

+1 1/2 loop over `Locations` in column
- **+1/2** correct loop (traverse entire column or until empty location found)
- **+1** construct `Location` object *in context of loop*
  - **+1/2** attempt using column
  - **+1/2** correct

+1 1/2 find drop `Location`
- **+1/2** check if constructed `Location` is empty
- **+1** if exists, return empty `Location` with largest row # (*no loop, no point*)

+1/2 return `null` if column is full

Part B: `dropMatchesNeighbors` **5 1/2 points**

+1 get drop `Location`
- **+1/2** attempt (must call `dropLocationForColumn`)
- **+1/2** correct (must use result)

+1/2 return `false` if drop location is `null`

+1 1/2 get neighboring pieces
- **+1/2** attempt to access adj. neighbors (`getNeighbor` or `neighborsOf` or row/column access)
- **+1/2** correctly access 3 E/W/S neighbor `Location` objects
- **+1/2** correctly access 3 neighbor `Piece` objects

+2 1/2 determine matches
- **+1/2** correct `null` neighbor test
- **+1** compare colors of pieces
  - **+1/2** attempt (must reference `pieceColor`)
  - **+1/2** correct
- **+1** return correct Boolean value1

Usage: -1 `environment` or missing `theEnv`

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. 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 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: Drop Game (MBS)

PART A:

```
public Location dropLocationForColumn(int column)
{
  for (int r = theEnv.numRows()-1; r >= 0; r--)
  {
    Location nextLoc = new Location(r, column);
    if (theEnv.isEmpty(nextLoc))
    {
      return nextLoc;
    }
  }
  return null;
}
```

ALTERNATE SOLUTION

```
public Location dropLocationForColumn(int column)
{
  int maxRow = -1;
  for (int r = 0; r < theEnv.numRows(); r++)
  {
    if (theEnv.isEmpty(new Location(r, column)))
    {
      maxRow = r;
    }
  }

  if (maxRow < 0)
  {
    return null;
  }
  return new Location(maxRow, column);
}
```

Question 4: Drop Game (MBS) (continued)

PART B:

```
public boolean dropMatchesNeighbors(int column, Color pieceColor)
{
  Location loc = dropLocationForColumn(column);
  if (loc == null)
  {
    return false;
  }
  Piece n1 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.WEST)));
  Piece n2 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.EAST)));
  Piece n3 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.SOUTH)));
  return (n1 != null && n1.color().equals(pieceColor) &&
          n2 != null && n2.color().equals(pieceColor) &&
          n3 != null && n3.color().equals(pieceColor));
}
```

ALTERNATE SOLUTION

```
public boolean dropMatchesNeighbors(int column, Color pieceColor)
{
  Location loc = dropLocationForColumn(column);
  if (loc == null)
  {
    return false;
  }
  ArrayList neighbors = theEnv.neighborsOf(loc);
  int colorCount = 0;
  for (int i = 0; i < neighbors.size(); i++)
  {
    Piece nextNbr = (Piece)(theEnv.objectAt((Location)neighbors.get(i)));
    if (nextNbr != null && nextNbr.color().equals(pieceColor))
    {
      colorCount++;
    }
  }

  return (colorCount == 3);
}
```

A4 A1

(a) Write the `DropGame` method `dropLocationForColumn`, which returns the resulting `Location` for a piece dropped into the specified column. If there are no empty locations in the column, the method should return `null`. Otherwise, of the empty locations in the column, the location with the largest row index should be returned.

In writing `dropLocationForColumn`, you may use any methods defined in the `DropGame` class or accessible methods of the case study classes.

Complete method `dropLocationForColumn` below.

```
// returns null if no empty locations in column;
// otherwise, returns the empty location with the
// largest row index within the specified column;
// precondition: 0 <= column < theEnv.numCols()
public Location dropLocationForColumn(int column)
{
   int row = 0;
   Location loc = new Location(row, column);
   if (!theEnv.isEmpty(loc))
      return null;
   while (theEnv.isEmpty(loc) && row < theEnv.numRows())
   {
      row++;
      loc = new Location(row, column);
   }
   row--;
   return new Location(row, column);
}
```

GO ON TO THE NEXT PAGE.

A4A2

(b) Write the `DropGame` method `dropMatchesNeighbors`, which returns `true` if dropping a piece of a given color into a specific column will match the color of three of its neighbors. The location to be checked for matches with its neighbors is the location identified by method `dropLocationForColumn`. If there are no empty locations in the column, `dropMatchesNeighbors` returns `false`.

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

Complete method `dropMatchesNeighbors` below.

```
// returns true if dropping a piece of the given color into the
// specified column matches color with three neighbors;
// otherwise, returns false
// precondition: 0 <= column < theEnv.numCols()
public boolean dropMatchesNeighbors(int column, Color pieceColor)
{
    Location loc = dropLocationForColumn(column);
    if (loc == null)
      return false;
    return((Piece) theEnv.objectAt(theEnv.getNeighbor(loc, Direction.WEST))).color().equals(
        pieceColor) && ((Piece) theEnv.objectAt(theEnv.getNeighbor(loc,
        Direction.EAST))).color().equals(pieceColor) && ((Piece) theEnv.objectAt(
        theEnv.getNeighbor(loc, Direction.SOUTH))).color-equals(pieceColor);
}
```

A4B,

(a) Write the `DropGame` method `dropLocationForColumn`, which returns the resulting `Location` for a piece dropped into the specified column. If there are no empty locations in the column, the method should return `null`. Otherwise, of the empty locations in the column, the location with the largest row index should be returned.

In writing `dropLocationForColumn`, you may use any methods defined in the `DropGame` class or accessible methods of the case study classes.

Complete method `dropLocationForColumn` below.

```
// returns null if no empty locations in column;
// otherwise, returns the empty location with the
// largest row index within the specified column;
// precondition: 0 <= column < theEnv.numCols()
public Location dropLocationForColumn(int column)
{  Location drop;
   for (int x = theEnv.numRows()-1; x >= 0; x--)
   {  if (theEnv.isEmpty(x, column))
       {  drop = Location (x, column);
            x = -1;
       }
       else if (x == 0)
           drop = null;
   }
   return drop;
}
```

A4 B2

(b) Write the DropGame method dropMatchesNeighbors, which returns true if dropping a piece of a given color into a specific column will match the color of three of its neighbors. The location to be checked for matches with its neighbors is the location identified by method dropLocationForColumn. If there are no empty locations in the column, dropMatchesNeighbors returns false.

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

Complete method dropMatchesNeighbors below.

```
// returns true if dropping a piece of the given color into the
// specified column matches color with three neighbors;
// otherwise, returns false
// precondition: 0 <= column < theEnv.numCols()
public boolean dropMatchesNeighbors(int column, Color pieceColor)
{
   if(dropLocationForColumn(column) !=null)
   {
   if( theEnv.getNeighborsOf( dropLocationForColumn(column)).get Color()==pieceColor)
     return true;
   else
     return false;
   }
   else
    return false;
}
```

GO ON TO THE NEXT PAGE.

A4C1

(a) Write the `DropGame` method `dropLocationForColumn`, which returns the resulting `Location` for a piece dropped into the specified column. If there are no empty locations in the column, the method should return `null`. Otherwise, of the empty locations in the column, the location with the largest row index should be returned.

In writing `dropLocationForColumn`, you may use any methods defined in the `DropGame` class or accessible methods of the case study classes.

Complete method `dropLocationForColumn` below.

```
// returns null if no empty locations in column;
// otherwise, returns the empty location with the
// largest row index within the specified column;
// precondition: 0 <= column < theEnv.numCols()
public Location dropLocationForColumn(int column)
{ int count; int i;
  int rows = theEnv.numRows();
  for (i=0 ; i < rows; i++) {
    if (isEmpty(i, column))
      count ++;
  }
  if (count == rows)
    return (Location)(rows, column);
  else
    return (Location) (rows - i, column);

  ~~if (objectAt(0, column)~~
  if((theEnv.objectAt(0, column) == false)
    return null;

}
```

A4C2

(b) Write the DropGame method `dropMatchesNeighbors`, which returns `true` if dropping a piece of a given color into a specific column will match the color of three of its neighbors. The location to be checked for matches with its neighbors is the location identified by method `dropLocationForColumn`. If there are no empty locations in the column, `dropMatchesNeighbors` returns `false`.

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

Complete method `dropMatchesNeighbors` below.

```
// returns true if dropping a piece of the given color into the
// specified column matches color with three neighbors;
// otherwise, returns false
// precondition: 0 <= column < theEnv.numCols()
public boolean dropMatchesNeighbors(int column, Color pieceColor)
```

```
{
  Location loc = dropLocationForColumn(column);
  if (loc.numAdjacentNeighbors() >= 3)  &&
     (theEnv.getNeighbor(loc, West)).isEmpty() == false
  && (theEnv.getNeighbor(loc, South)).isEmpty() == false)
  && (theEnv.getNeighbor(loc, East)).isEmpty() == false)
 && (theEnv.neighborsOf(loc).color() == WHITE)
   { return true; }
 else return false;

}
```

AP® COMPUTER SCIENCE A
2006 SCORING COMMENTARY

Question 4

Overview

This question was based on the Marine Biology Simulation (MBS) case study and focused on abstraction and code reuse. Students needed to show their understanding of the case study and its interacting classes in order to implement methods for a particular board game. A `Piece` class was provided for representing game pieces and implemented the `Locatable` interface. The `DropGame` class then represented the board as an `Environment of Pieces`. In part (a) students were required to complete the `dropLocationForColumn` method, which found the `Location` in that column where a dropped piece would rest. This involved traversing the column and identifying the empty location with highest row index. In part (b) students were required to complete the `dropMatchesNeighbors` method, which determined whether a piece dropped in a specified column would result in a win. This involved first identifying the drop location for that `column` (by calling the `dropLocationForColumn` method from part (a)) and then checking neighboring locations for pieces of the same color. This last step could be accomplished in a variety of ways using `Environment`, `Location`, and `Direction` methods.

Sample: A4A
Score: 9

Part (a): The code correctly sets a counter `row` to 0 and a `Location`, `loc` to the top element of the column. This top element is examined, and if the location is empty, `null` is returned. If the top item is empty, the code continues by processing a loop that continues as long as there are empty locations available. It is important to notice that the loop body always advances the row counter and the corresponding location variable so that when the loop terminates, the location `loc` is actually one row further down than it needs to be. This is not a problem, even when the column is entirely empty. In the empty column case, the location `loc` will eventually reach the `theEnv.numRows()` position, which is outside of the environment. Although this may appear problematic, the loop is still correct since `isEmpty()` returns `false` when an invalid location is passed to it. After exiting the loop, the code compensates for going one position too far by reducing the row counter by one and correctly returning the corresponding location within the row. The student response is correct and earned the 3½ points for part (a).

Part (b): The student correctly obtains and saves the drop location through a valid call on `dropLocationForColumn` using the appropriate column. Once the call is made, the returned location is checked to make sure that it is a valid location. If it is not valid, there can be no match, and the code correctly returns `false`. Otherwise, the code returns the value of a complex Boolean expression that determines whether or not the `pieceColor` matches the colors of the neighboring pieces. The Boolean expression correctly finds the neighbors through valid calls on `getNeighbor`. However, prior to obtaining the objects at the retrieved locations, the code fails to check to see if the location is `null`. A ½ point was deducted for not making this check. Despite this, the code correctly uses `objectAt` to obtain the piece, correctly accesses the color of the piece, and makes valid comparisons to `pieceColor`. The expression returns `true` when all neighbors match the `pieceColor` and `false` otherwise. Aside from the failure to check to see if the piece locations are `null`, the code is correct and earned 5 points for part (b). The total score for this student was 8½ points, which (using program policy) was rounded to a 9 for the final score.

Sample: A4B
Score: 5

Part (a): The student starts by looping from the bottom to the top of the column. The loop correctly accesses each row of the column. There is an attempt to determine whether the location under consideration is empty. However, although the form of the call on `isEmpty` is correct, the parameter is merely a row/column pair that suggests a location. A ½ point was deducted for failing to correctly check that the location is empty. If the location is empty, the code attempts to construct the correct empty location, but fails to use `new` to create the `Location` object. The student received credit for attempting to create a location but lost a ½ point because it is not created correctly. After saving the drop location, the code proceeds to set the loop control variable `x` equal to a -1. This has the effect of terminating the loop upon the start of the next iteration. If the location is occupied the `else` branch is taken where the code further checks to see if it is at the top location in the column under consideration. If it is under consideration the column is full and the drop location is accordingly saved as `null`. Upon loop termination the saved drop location is returned correctly. The response earned 2½ points for part (a).

Part (b): The student correctly obtains the drop location through a valid call on `dropLocationForColumn` using the appropriate column. The code that attempts consideration of neighboring pieces is protected from considering a `null` drop location through a well constructed `if` statement that correctly returns `false` when the drop location is `null`. However, although an attempt is made to consider neighboring pieces through a call on a nonexistent method `getNeighborsOf`, there is no sense in the code of considering each location individually. Therefore, the score was reduced by 1 point (a ½ point for incorrectly accessing locations and a ½ point for incorrectly accessing pieces). Further, no attempt is made to see if an examined location is `null` or not, which led to a further ½ point deduction. Despite this, there is an attempt to make a color comparison. It is incorrect because the `getColor` method is not defined for pieces. Failure to make the color check correctly reduced the score by an additional ½ point. Since the code does not consider individual neighbor pieces it cannot correctly determine whether or not a piece dropped in the drop location will win. This led to another 1 point deduction. The response earned 2½ points for part (b).

Sample: A4C
Score: 2

Part (a): The student starts by looping from the top to the bottom of the column. The loop correctly accesses each row of the column. There is an attempt to determine whether a place under consideration is empty. However, a row/column pair by itself is insufficient to show that there is an attempt to create a location in the context of a loop. Consequently, the score was reduced by 1 point (a ½ point for the attempt to create a location in the context of the loop and a ½ point for the correct creation of location). Likewise, in the absence of an attempt to create a location, the method `isEmpty` does not receive a correct parameter. By itself, this error reduced the score by a ½ point. It is worth noting that failing to refer to `theEnv` when calling the method is also erroneous, but the credit for this ½ point was already deducted due to the parameter error, so no further reduction was taken. Although the variable `count` is calculated within the loop and is referenced in the `if` statement following the loop, the attempt to create locations in the `if-else` statement does not refer to a row/column pair that is related to `count`. Without the context of referring to a location within the loop, no credit could be awarded for a correct return. One point was deducted for the incorrect return. The `if-else` statement also has the effect of terminating the method before consideration of the final `if` statement. Since there is a `return` in each branch of the `if-else` statement, the final `if` statement will never be executed. Therefore, no credit can be earned for returning `null` when the column is full, and a ½ point was deducted for this error. This response earned a ½ point for part (a).

Question 4 (continued)

Part (b): The student correctly calls the method `dropLocationForColumn()` using the appropriate column. The student earned both the attempt and correctness ½ points for a total of 1 point. However, the code that attempts to consider neighboring pieces is not protected from considering a `null` drop location. A ½ point was deducted for failing to test for a `null` drop location. The `if` statement that follows the method call incorrectly attempts to access the `numAdjacentNeighbors()` method from `loc` instead of `theEnv`. While there is an attempt to examine appropriate neighbor pieces of the drop location, it fails since the constants `WEST`, `SOUTH`, and `EAST` are accessed incorrectly (they must be `Direction.WEST` and so on). The neighbor locations are not accessed correctly which results in a ½ point deduction. Further, there is no correct access of any piece object at any of the locations. This resulted in a further deduction of a ½ point. There is a reference to color, but it is not related to a single object or a single location. Further, there is no color comparison to the `pieceColor` parameter. Failing to attempt a comparison between `pieceColor` and each neighbor's color resulted in a deduction of 1 point (the attempt ½ point and the correct ½ point). Attempting to find out if the locations are empty through the use of the `isEmpty` method is not sufficient to protect the code from `null` locations. Note that the `getNeighbor` method returns the adjacent neighboring location in the specified direction—whether it is valid or invalid. The method `isEmpty` returns `false` when the location is invalid. Therefore, this is not an appropriate `null` test and a ½ point was deducted accordingly. Since the code relies on erroneously accessing an aggregate color of all of the neighbors, the return value is not correct. This resulted in a deduction of 1 point for incorrect return. It is worth noting that even if the individual piece colors are accessed properly, the resulting code is still incorrect due to a comparison of the obtained color to `WHITE` instead of the `pieceColor`. This response earned 1½ points for part (b).