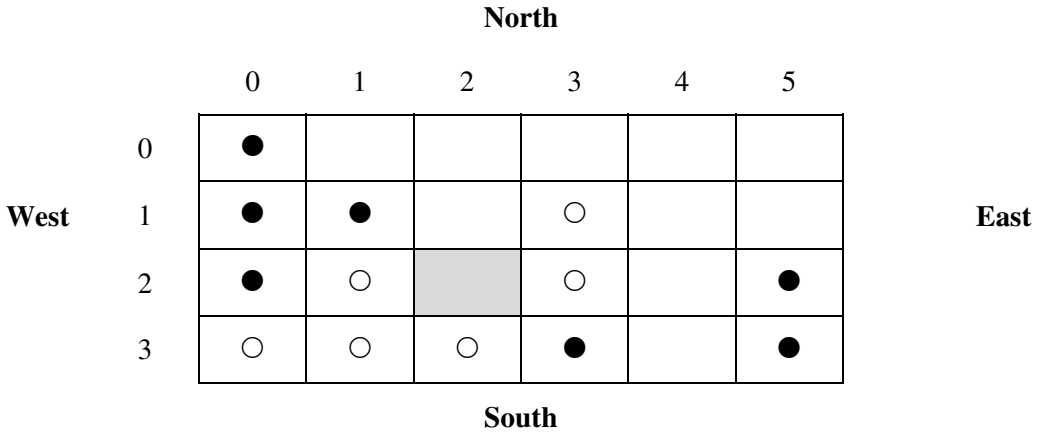
4. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

   Consider using the `BoundedEnv` class from the Marine Biology Simulation case study to model a game board. In this implementation of the `Environment` interface, each location has at most **four** neighbors. Those neighbors are determined by the `Environment` method `neighborsOf`.

   DropGame is a two-player game that is played on a rectangular board. The players — designated as BLACK and WHITE — alternate, taking turns dropping a colored piece in a column. A dropped piece will fall down the chosen column until it comes to rest in the empty location with the largest row index. If the location for the **newly dropped** piece has **three** neighbors that match its color, the player that dropped this piece wins the game.

   The diagram below shows a sample game board on which several moves have been made.



   The following chart shows where a piece dropped in each column would land on this board.

Column	Location for Piece Dropped in the Column
0	No piece can be placed, since the column is full
1	(0, 1)
2	(2, 2)
3	(0, 3)
4	(3, 4)
5	(1, 5)

   Note that a WHITE piece dropped in column 2 would land in the shaded cell at location (2, 2) and result in a win for WHITE because the three neighboring locations — (2, 1), (3, 2), and (2, 3) — contain WHITE pieces. This move is the only available winning move on the above game board. Note that a BLACK piece dropped in column 1 would land in location (0, 1) and not result in a win because the neighboring location (0, 2) does not contain a BLACK piece.

The `Piece` class implements the `Locatable` interface and is defined as follows.

```
public class Piece implements Locatable
{
  // returns location of this Piece
  public Location location()
  {  /* implementation not shown */  }

  // returns color of this Piece
  public Color color()
  {  /* implementation not shown */  }

  // There may be fields, constructors, and methods that are not shown.
}
```

An incomplete definition of the `DropGame` class is shown below. The class contains a private instance variable `theEnv` to refer to the `Environment` that represents the game board. Players will add `Piece` objects to this environment as they take turns. You will implement two methods for the `DropGame` class.

```
public class DropGame
{
  private Environment theEnv;  // contains Piece objects


  // 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)
  {  /* to be implemented in part (a) */  }


  // 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)
  {  /* to be implemented in part (b) */  }


  // There may be fields, constructors, and methods that are not shown.
}
```

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.*

AP® COMPUTER SCIENCE A
2006 CANONICAL SOLUTIONS

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);
}
```