AP® Computer Science
GridWorld Case Study

Sample Exam Questions

Multiple-choice Questions

1. A `FastCritter` is to behave exactly like a `Critter`, except that it can move up to two grid locations in each direction (assuming that it is not blocked by another actor). To achieve this behavior, which method should be overridden?

   (A) `getActors`

   (B) `processActors`

   (C) `getMoveLocations`

   (D) `selectMoveLocation`

   (E) `makeMove`

2. Consider the following code segment.

```
Grid<Actor> grid = new BoundedGrid<Actor>(10, 10);

Bug bug1 = new Bug();
bug1.setDirection(Location.EAST);
bug1.putSelfInGrid(grid, new Location(3, 5));

Bug bug2 = new Bug();
bug2.setDirection(Location.SOUTH);
bug2.putSelfInGrid(grid, new Location(2, 6));

bug1.act();
bug2.act();
```

   What is the result of executing the code segment?

   (A) `bug1` and `bug2` will both occupy location (3, 6).

   (B) `bug1` will move into location (3, 6) and `bug2` will turn.

   (C) `bug1` will move into location (3, 6) and `bug2` will do nothing.

   (D) `bug1` will move into location (3, 6) and `bug2` will throw an exception.

   (E) `bug1` will move into location (3, 6) but will be removed when `bug2` moves into the same location, (3, 6).

3. Consider the following code segment.

```
Grid<Actor> grid = new BoundedGrid<Actor>(10, 10);
Bug ladybug = new Bug();
ladybug.putSelfInGrid(grid, new Location(2, 8));
ladybug.setDirection(Location.WEST);
ladybug.act();
```

Under which of the following circumstances can `ladybug` advance to location (2,7) ?

I. The location (2,7) contains another `Bug`.

II. The location (2,7) contains a `Flower`.

III. The location (2,7) contains a `Rock`.

(A) I only

(B) II only

(C) III only

(D) I and II only

(E) I, II, and III

4. A `RockingCritter` replaces each adjacent `Flower` with a `Rock` and then moves like a regular `Critter`. The following three implementations have been proposed.

I.
```
public class RockingCritter extends Critter
{
  public ArrayList<Actor> getActors()
  {
    ArrayList<Actor> actors = new ArrayList<Actor>();
    for (Actor a : getGrid().getNeighbors(getLocation()))
    {
      if (a instanceof Flower)
        actors.add(a);
    }
    return actors;
  }

  public void processActors(ArrayList<Actor> actors)
  {
    for (Actor a : actors)
    {
      Location loc = a.getLocation();
      a.removeSelfFromGrid();
      Rock r = new Rock();
      r.putSelfInGrid(getGrid(), loc);
    }
  }
}
```

II.
```
public class RockingCritter extends Critter
{
  public void processActors(ArrayList<Actor> actors)
  {
    for (Actor a : actors)
    {
      if (a instanceof Flower)
      {
        Location loc = a.getLocation();
        a.removeSelfFromGrid();
        Rock r = new Rock();
        r.putSelfInGrid(getGrid(), loc);
      }
    }
  }
}
```

III.
```
public class RockingCritter extends Critter
{
  public void makeMove(Location loc)
  {
    for (Actor a : getGrid().getNeighbors(getLocation()))
    {
      if (a instanceof Flower)
      {
        Location rLoc = a.getLocation();
        a.removeSelfFromGrid();
        Rock r = new Rock();
        r.putSelfInGrid(getGrid(), rLoc);
      }
    }

    super.makeMove(loc);
  }
}
```

Which of the implementations is (are) correct?

(A) I only

(B) II only

(C) III only

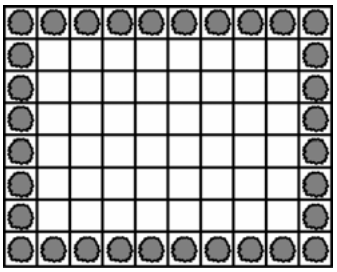
(D) I and II only

(E) I, II, and III

5. Consider the following instance variable and method.

```
private BoundedGrid<Actor> grid;

public void placeRock(int row, int col)
{
  new Rock().putSelfInGrid(grid, new Location(row, col));
}
```

Consider the problem of marking the borders of the grid with rocks as shown in the following picture.



The following code segments are proposed as solutions to the problem.

I.
```
for (int j = 0; j < grid.getNumRows(); j++)
{
  placeRock(j, 0);
  placeRock(j, grid.getNumCols() - 1);
}
for (int k = 0; k < grid.getNumCols(); k++)
{
  placeRock(0, k);
  placeRock(grid.getNumRows() - 1, k);
}
```

II.
```
for (int j = 0; j < grid.getNumRows() - 1; j++)
{
  placeRock(j, 0);
  placeRock(j + 1, grid.getNumCols() - 1);
}
for (int k = 0; k < grid.getNumCols() - 1; k++)
{
  placeRock(0, k + 1);
  placeRock(grid.getNumRows() - 1, k);
}
```

III.
```
for (int j = 0; j < grid.getNumRows(); j++)
{
  for (int k = 0; k < grid.getNumCols(); k++)
  {
    Location loc = new Location(j, k);
    if (grid.getValidAdjacentLocations(loc).size() < 8)
      placeRock(j, k);
  }
}
```

Which of the code segments will correctly place rocks along the borders of the grid?

(A) I only

(B) II only

(C) III only

(D) I and III only

(E) I, II, and III

Answers to Multiple-choice Questions

1. C
2. B
3. B
4. D
5. E

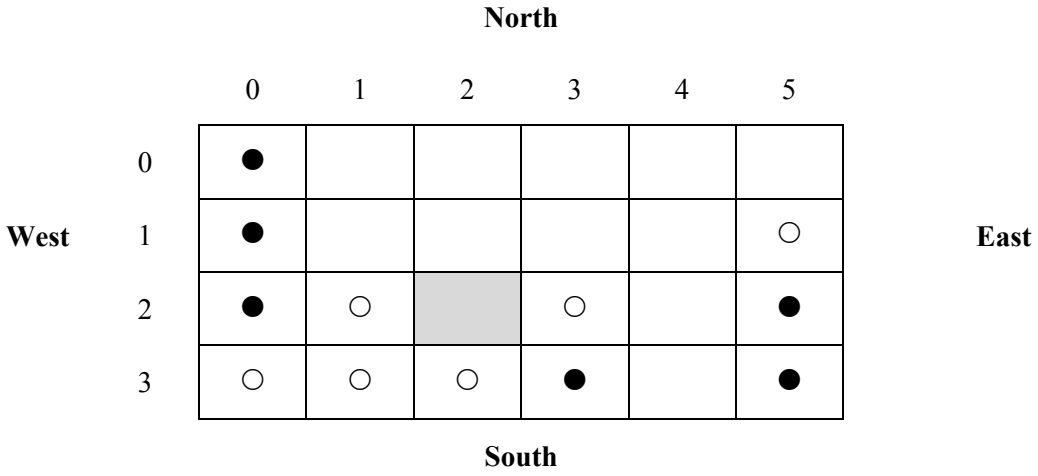
Free-response Questions

1. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

   Consider using the `BoundedGrid` class from the GridWorld case study to model a game board.

   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 **at least four** 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	(1, 1)
2	(2, 2)
3	(1, 3)
4	(3, 4)
5	(0, 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 four neighboring locations — (2, 1), (3, 1), (3, 2), and (2, 3) — contain WHITE pieces. This move is the only available winning move on the above game board.

The `Piece` class is defined as follows.

```
public class Piece
{
  /** @return the color of this Piece
   */
  public Color getColor()
  {  /* implementation not shown */  }

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

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

```
public class DropGame
{
  private Grid<Piece> theGrid;


  /** @param column a column index in the grid
   *          Precondition: 0 ≤ column < theGrid.getNumCols()
   *  @return null if no empty locations in column;
   *          otherwise, the empty location with the largest row index within column
   */
  public Location dropLocationForColumn(int column)
  {  /* to be implemented in part (a) */  }


  /** @param column a column index in the grid
   *          Precondition: 0 ≤ column < theGrid.getNumCols()
   *  @param pieceColor the color of the piece to be dropped
   *  @return true if dropping a piece of the given color into the specified column matches color
   *                with at least four neighbors;
   *          false otherwise
   */
  public boolean dropMatchesNeighbors(int column, Color pieceColor)
  {  /* to be implemented in part (b) */  }


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

(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.

Complete method `dropLocationForColumn` below.

```
/** @param column a column index in the grid
 *         Precondition: 0 ≤ column < theGrid.getNumCols()
 *  @return null if no empty locations in column;
 *          otherwise, the empty location with the largest row index within column
 */
public Location dropLocationForColumn(int column)
```

(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 at least four 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`.

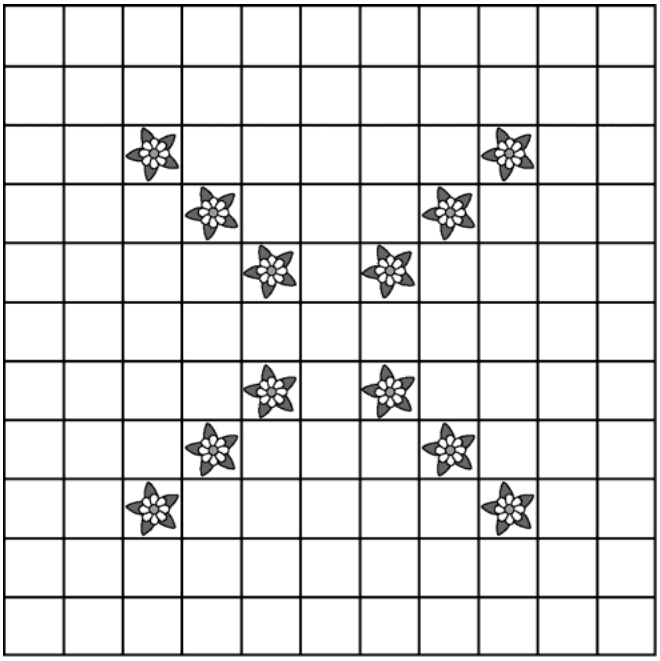
In writing `dropMatchesNeighbors`, you may assume that `dropLocationForColumn` works as specified regardless of what you wrote in part (a).

Complete method `dropMatchesNeighbors` below.

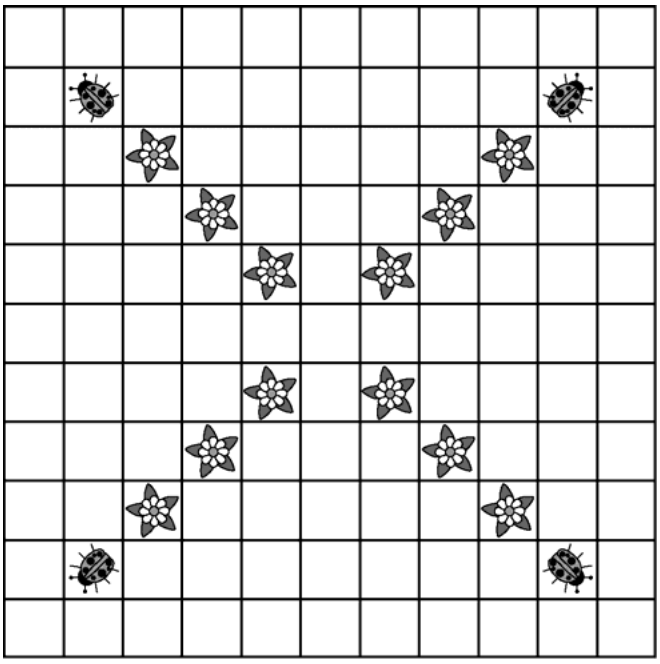
```
/** @param column a column index in the grid
 *         Precondition: 0 ≤ column < theGrid.getNumCols()
 *  @param pieceColor the color of the piece to be dropped
 *  @return true if dropping a piece of the given color into the specified column matches color
 *                with at least four neighbors;
 *          false otherwise
 */
public boolean dropMatchesNeighbors(int column, Color pieceColor)
```

2. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

   In this question, you will consider two approaches for implementing the design of a bug that produces an X-shaped pattern of flowers. You may assume that there are no other actors in the grid and that there is enough room for the X to be placed in the grid with a row of empty locations surrounding the area filled by the X. Here is a pattern in which each arm of the X has length 3. Note that the center of the X is not marked with a flower.



(a) In the first approach, the bug releases four helper bugs that each drop the appropriate number of flowers along one arm of the X.



This approach is implemented by a class `XBug1`. The declaration of the `XBug1` class is as follows. The `act` method puts four instances of a class `LineBug` (which you will need to implement) into the grid and then removes itself.

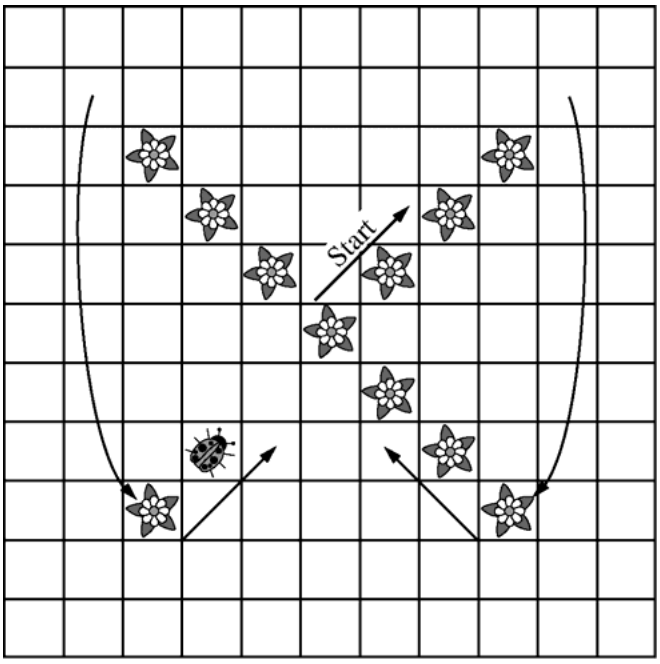
```
public class XBug1 extends Bug
{
  private int length; // the length of each of the arms of the X

  public XBug1(int aLength)
  {  length = aLength;  }

  public void act()
  {
    Grid<Actor> gr = getGrid();
    Location loc = getLocation();
    int dir = Location.NORTHEAST;
    for (int k = 0; k < 4; k++)
    {
      LineBug lbug = new LineBug(length);
      lbug.setDirection(dir);
      lbug.putSelfInGrid(gr, loc.getAdjacentLocation(dir));
      dir += Location.RIGHT;
    }
    removeSelfFromGrid();
  }
}
```

Write the declaration for a class `LineBug` with the following features:

- A `LineBug` is constructed with an integer parameter, denoting the number of flowers that the bug drops during its lifetime.
- When the `act` method is called, if the appropriate number of flowers has already been dropped, the `LineBug` removes itself from the grid; otherwise, the `LineBug` moves once, thereby dropping a flower.

Write the complete `LineBug` class, including all instance variables, a constructor, and any required methods.

(b) In the second approach, the bug drops flowers along the path in successive calls to `act`. When the bug has reached the end of an arm, it jumps to the end of another arm, as shown below.



The declaration of the `XBug2` class is as follows.

```
public class XBug2 extends Bug
{
  private int length;             // the length of each of the arms of the X
  private int steps;              // the number of times the act method has been called
  private Location bottomLeft;    // the location of the bottom left end of the X
  private Location bottomRight;   // the location of the bottom right end of the X

  public XBug2(int aLength)
  {
    length = aLength;
    steps = 0;
  }

  public void putSelfInGrid(Grid<Actor> gr, Location loc)
  {
    /* puts the bug in the grid and initializes the bottomLeft and bottomRight locations */
  }

  public void act()
  {   /* to be implemented in part (b) */   }
}
```

Write the `XBug2 act` method. You may assume that the instance variables have been initialized prior to the first call of the `act` method.

In each call to the `act` method, the `XBug2` makes one call to `move`. It starts in the center point of the X and moves northeast. When it reaches the top right end of the X, it calls `moveTo` to move to the bottom right end of the X. It then moves northwest. When it reaches the top left end of the X, it calls `moveTo` to move to the bottom left end of the X. When the X pattern is completed, the `XBug2` removes itself from the grid in the next call to the `act` method.

Complete method `act` below.

```
public void act()
```

Solutions to Free-response Questions

1.
Part (a)

```
/** @param column  a column index in the grid
 *          Precondition: 0 ≤ column < theGrid.getNumCols()
 *  @return null  if no empty locations in  column;
 *           otherwise, the empty location with the largest row index within  column
 */
public Location dropLocationForColumn(int column)
{
  for (int r = theGrid.getNumRows() - 1; r >= 0; r--)
  {
    Location nextLoc = new Location(r, column);
    if (theGrid.get(nextLoc) == null)
    {
      return nextLoc;
    }
  }
  return null;
}
```

Part (b)

```
/** @param column  a column index in the grid
 *          Precondition: 0 ≤ column < theGrid.getNumCols()
 *  @param pieceColor  the color of the piece to be dropped
 *  @return true  if dropping a piece of the given color into the specified column matches color
 *                with at least four neighbors;
 *          false  otherwise
 */
public boolean dropMatchesNeighbors(int column, Color pieceColor)
{
  Location loc = dropLocationForColumn(column);
  if (loc == null)
  {
    return false;
  }

  ArrayList<Piece> neighbors = theGrid.getNeighbors(loc);
  int colorCount = 0;
  for (Piece nextNbr : neighbors)
  {
    if (nextNbr.getColor().equals(pieceColor))
    {
      colorCount++;
    }
  }

  return (colorCount >= 4);
}
```

2.
Part (a)

```
import info.gridworld.actor.Bug;

/**
 * This bug traces a line segment of a given length and then removes itself.
 */
public class LineBug extends Bug
{
  private int length;

  /**
   * Constructs a line bug with a given length
   * @param aLength the length of the line segment that this bug traces
   */
  public LineBug(int aLength)
  {
    length = aLength;
  }

  public void act()
  {
    if (length > 0)
    {
      move();
      length--;
    }
    else
      removeSelfFromGrid();
  }
}
```

Part (b)

```
public void act()
{
  if (steps == 0)
  {
    setDirection(Location.NORTHEAST);
  }
  else if (steps == length + 1)
  {
    moveTo(bottomRight);
    setDirection(Location.NORTHWEST);
  }
  else if (steps == 3 * length + 2)
  {
    moveTo(bottomLeft);
    setDirection(Location.NORTHEAST);
  }
  else if (steps >= 4 * length + 2)
  {
    removeSelfFromGrid();
  }

  move();
  steps++;
}
```