AP® COMPUTER SCIENCE A
2008 SCORING GUIDELINES

Question 3: Opossum Critter (GridWorld)

Part A: processActors **6 points**

+1/2 initialize friend/foe counter(s)

+2 1/2 loop and identify actors
- **+1** traverse `actors`
  - **+1/2** correctly access an element of `actors` (in context of loop)
  - **+1/2** access all elements of `actors` (lose this if index out-of-bounds)
- **+1 1/2** identify actor category and update counters (in context of loop)
  - **+1/2** call `isFriend(`*nextActorFromList*`)`
  - **+1/2** call `isFoe(`*nextActorFromList*`)`
  - **+1/2** update counters appropriately in both cases

+3 update `OpossumCritter` state
- **+1** correctly identify whether to play dead
- **+1** appropriate result if playing dead
  - **+1/2** `setColor(Color.BLACK)`
  - **+1/2** `numStepsDead++`
- **+1** appropriate result if normal
  - **+1/2** `setColor(Color.ORANGE)`
  - **+1/2** `numStepsDead = 0`

Part B: selectMoveLocation **3 points**

+1 determine appropriate case (using == with `Color` is okay)
- **+1/2** correctly identify one case (dead, playing dead, normal)
- **+1/2** correctly identify all three cases

+2 appropriate return values
- **+1/2** return null if really dead
- **+1/2** return current location if playing dead
- **+1** return `super.selectMoveLocation(locs)` otherwise
  - **+1/2** `super.selectMoveLocation(locs)`
  - **+1/2** return value from call

Usage:
- -1 if violate postconditions (e.g., `removeSelfFromGrid()`)
- -1 for BLACK or "Black" instead of `Color.BLACK`
- -1/2 for call to (nonexistent) default `Location` constructor

AP® COMPUTER SCIENCE A
2008 CANONICAL SOLUTIONS

Question 3: Opossum Critter (GridWorld)

PART A:

```
public void processActors(ArrayList<Actor> actors)
{
  int numFriends = 0;
  int numFoes = 0;

  for (Actor nextActor : actors)
  {
    if (isFriend(nextActor))
      numFriends++;
    else if (isFoe(nextActor))
      numFoes++;
  }

  if (numFoes > numFriends)
  {
    setColor(Color.BLACK);
    numStepsDead++;
  }
  else
  {
    setColor(Color.ORANGE);
    numStepsDead = 0;
  }
}
```

PART B:

```
public Location selectMoveLocation(ArrayList<Location> locs)
{
  if (numStepsDead == 3)
    return null;
  else if (numStepsDead > 0)
    return getLocation();
  else
    return super.selectMoveLocation(locs);
}
```

OR

```
public Location selectMoveLocation(ArrayList<Location> locs)
{
  if (getColor().equals(Color.BLACK))
  {
    if (numStepsDead == 3)
      return null;
    else
      return getLocation();
  }
  return super.selectMoveLocation(locs);
}
```

A3a.

(a) Override the `processActors` method for the `OpossumCritter` class. This method should look at all elements of `actors` and determine whether or not to play dead according to the types of the actors. If there are more foes than friends, the `OpossumCritter` indicates that it is playing dead by changing its color to `Color.BLACK`. When not playing dead, it sets its color to `Color.ORANGE`. The instance variable `numStepsDead` should be updated to reflect the number of consecutive steps the `OpossumCritter` has played dead.

Complete method `processActors` below.

```
/** Whenever actors contains more foes than friends, this OpossumCritter plays dead.
 *  Postcondition: (1) The state of all actors in the grid other than this critter and the
 *  elements of actors is unchanged. (2) The location of this critter is unchanged.
 *  @param actors a group of actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
int friendCount=0;
int foeCount=0;

for(Actor a: actors)
{
  if(isFriend(a)=true)
  {
    friendcount++;
  }
  else
  {
    foeCount++;
  }

if(foecount>friendcount)
   setColor(Color.BLACK); numStepsDead++;
else
   setColor(Color.ORANGE);
   numStepsDead=0;
```

Part (b) begins on page 14.

A3a 2

(b) Override the `selectMoveLocation` method for the `OpossumCritter` class. When the `OpossumCritter` is not playing dead, it behaves like a `Critter`. The next location for an `OpossumCritter` that has been playing dead for three consecutive steps is `null`. Otherwise, an `OpossumCritter` that is playing dead remains in its current location.

Complete method `selectMoveLocation` below.

```
/** Selects the location for the next move.
 *  Postcondition: (1) The returned location is an element of locs, this critter's current location,
 *  or null. (2) The state of all actors is unchanged.
 *  @param locs the possible locations for the next move
 *  @return the location that was selected for the next move, or null to indicate
 *          that this OpossumCritter should be removed from the grid.
 */
public Location selectMoveLocation(ArrayList<Location> locs)
```

```
if(numStepsDead == 0)
{
  return super.selectMoveLocation(loc);
}
else if(numStepsDead == 3)
{
  returnNull
}
else
{
  return getLocation();
```

GO ON TO THE NEXT PAGE.

A3b,

(a) Override the `processActors` method for the `OpossumCritter` class. This method should look at all elements of `actors` and determine whether or not to play dead according to the types of the actors. If there are more foes than friends, the `OpossumCritter` indicates that it is playing dead by changing its color to `Color.BLACK`. When not playing dead, it sets its color to `Color.ORANGE`. The instance variable `numStepsDead` should be updated to reflect the number of consecutive steps the `OpossumCritter` has played dead.

Complete method `processActors` below.

```
/** Whenever actors contains more foes than friends, this OpossumCritter plays dead.
 *  Postcondition: (1) The state of all actors in the grid other than this critter and the
 *  elements of actors is unchanged. (2) The location of this critter is unchanged.
 *  @param actors a group of actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
{
    int n= actors.size();
    if(n==0)
    {
       numStepsDead=0;
       return;
    }
    int friend=0;
    int foe=0;
    for(int x=0; x<n; x++)
    {
       if(actors.get(x).isFriend())
       { friend++;}
       if(actors.get(x).isFoe())
       { foe++;}
    }
    if(foe>friend)
    {
       setColor(Color.BLACK);
       setLocation(getLocation());
       numStepsDead++;
    }
    else
    {
       setColor(Color.ORANGE);
       numStepsDead=0;
    }
}
```

Part (b) begins on page 14.

GO ON TO THE NEXT PAGE.

A3b 2

(b) Override the `selectMoveLocation` method for the `OpossumCritter` class. When the `OpossumCritter` is not playing dead, it behaves like a `Critter`. The next location for an `OpossumCritter` that has been playing dead for three consecutive steps is `null`. Otherwise, an `OpossumCritter` that is playing dead remains in its current location.

Complete method `selectMoveLocation` below.

```
/** Selects the location for the next move.
 *  Postcondition: (1) The returned location is an element of locs, this critter's current location,
 *  or null. (2) The state of all actors is unchanged.
 *  @param locs the possible locations for the next move
 *  @return the location that was selected for the next move, or null to indicate
 *          that this OpossumCritter should be removed from the grid.
 */
public Location selectMoveLocation(ArrayList<Location> locs)
{
   int n = locs.size();
   if(n==0)
   {
      return getLocation();
   }
   if(numStepsDead>0 && numStepsDead<3)
   {
      return getLocation();
   }
   if(numStepsDead>=3)
   {
      return null;
   }
   int r = (int) (Math.random() * n);
   return locs.get(r);
}
```

GO ON TO THE NEXT PAGE.

A3c,

(a) Override the `processActors` method for the `OpossumCritter` class. This method should look at all elements of `actors` and determine whether or not to play dead according to the types of the actors. If there are more foes than friends, the `OpossumCritter` indicates that it is playing dead by changing its color to `Color.BLACK`. When not playing dead, it sets its color to `Color.ORANGE`. The instance variable `numStepsDead` should be updated to reflect the number of consecutive steps the `OpossumCritter` has played dead.

Complete method `processActors` below.

```
/** Whenever actors contains more foes than friends, this OpossumCritter plays dead.
 *  Postcondition: (1) The state of all actors in the grid other than this critter and the
 *  elements of actors is unchanged. (2) The location of this critter is unchanged.
 *  @param actors a group of actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
{
    Color Jill = new Color(Color.ORANGE);
    for (Actor a : actors)
      { if (!(a isFriend()))
          Color Bob = new Color(Color.BLACK);
          numStepsDead++;
      }
}
```

Part (b) begins on page 14.

A3c 2

(b) Override the `selectMoveLocation` method for the `OpossumCritter` class. When the `OpossumCritter` is not playing dead, it behaves like a `Critter`. The next location for an `OpossumCritter` that has been playing dead for three consecutive steps is `null`. Otherwise, an `OpossumCritter` that is playing dead remains in its current location.

Complete method `selectMoveLocation` below.

```
/** Selects the location for the next move.
 *  Postcondition: (1) The returned location is an element of locs, this critter's current location,
 *  or null. (2) The state of all actors is unchanged.
 *  @param locs the possible locations for the next move
 *  @return the location that was selected for the next move, or null to indicate
 *          that this OpossumCritter should be removed from the grid.
 */
public Location selectMoveLocation(ArrayList<Location> locs)
{
    if(numStepsDead >= 3)
      return null;
    super.selectMoveLocation();
}
```

GO ON TO THE NEXT PAGE.

AP® COMPUTER SCIENCE A
2008 SCORING COMMENTARY

Question 3

Overview

This question was based on the GridWorld case study and focused on abstraction and inheritance. Students showed their understanding of the case study and its interacting classes by extending `Critter` to derive an `OpossumCritter` class with modified behavior. In part (a) students were required to override the `processActors` method so that the surrounding neighbors were accessed and the state of the `OpossumCritter` updated according to the characteristics of those neighbors. In part (b) students had to override the `selectMoveLocation` method so that the resulting move (and ultimate survival of the `OpossumCritter`) depended upon its updated state.

Sample: A3a
Score: 8

In part (a) the student correctly initializes the two counter variables `friendcount` and `foecount`. The correct use of the for-each loop earned the access ½ point and the traverse-all ½ point. The method `isFriend` is correctly called, but the student lost ½ point for not calling `isFoe` and also lost ½ point for updating counters because the value of `foecount` will be wrong. The `if` statement following the loop correctly determines if this `OpossumCritter` should play dead by checking if the value of `foecount` is greater than the value of `friendcount`. If the critter is to play dead, the color is correctly set to black and `numStepsDead` is correctly incremented. Otherwise, the color is set to orange and `numStepsDead` is reset to 0. The student earned a total of 5 points for part (a).

In part (b) the student correctly identifies the three possible cases: if `numStepsDead` is 0 (should act like normal critter), if `numStepsDead` is 3 (dead), and otherwise (playing dead). If the critter is not threatened (`numStepsDead` = 0), it correctly calls `super.selectMoveLocation(locs)` and returns the result of this call. If the critter is dead, `null` is returned. If the critter is playing dead, it will not move so its current location is returned. The student earned a total of 3 points for part (b).

Sample: A3b
Score: 6

In part (a) the student first checks if `actors` has any elements in it (checking if the critter has anything that could be a considered a friend or a foe). If `actors.size()` is 0 (critter not threatened), the student resets `numStepsDead` to 0 but fails to set the critter's color to orange. The student earned ½ point for correctly initializing the two counter variables `friend` and `foe`. Since there is a `for` loop that correctly accesses an element of `actors` (`actors.get(x)`) and correctly traverses through all elements of `actors`, the student earned those two ½ points. The methods `isFriend` and `isFoe` are not correctly called, so the student lost these two ½ points. But the appropriate counters are updated in both the friend case and the foe case, so the student earned that ½ point.

The `if` statement following the loop correctly determines if this `OpossumCritter` should play dead by checking if the value of `foe` is greater than the value of `friend`. If the critter is to play dead, the color is correctly set to black and `numStepsDead` is correctly incremented. There is no deduction for the `setLocation` call because it does not change the critter's location. The student will lose the ½ point for changing the color to orange because of the failure to set the color in the not-threatened case at the beginning of this method. The student earned the ½ point for resetting `numStepsDead` to 0. The student earned a total of 4½ points for part (a).

Question 3 (continued)

In part (b) the student correctly identifies the three possible cases: `numstepsDead > 0 && numStepsDead <` 3 (playing dead), `numStepsDead >=` 3 (dead), and otherwise (should act like normal critter). By checking `locs.size()` in the beginning, the student is assuming that if there is no place to move, the critter should remain in its current location. This is not necessarily true. If `numStepsDead` = 3, the critter will have to die so `null` should be returned. The student lost the ½ `null` return point because of this. If the critter is playing dead, its current location is correctly returned so the student earned this ½ point. The student lost the two ½ points for not calling `super.selectMoveLocation(locs)` and not returning the result of this call. The student earned a total of 1½ points for part (b).

Sample: A3c
Score: 3

In part (a) the student does not initialize any counter variables, so that ½ point was lost. The correct use of the for-each loop earned the access ½ point and the traverse-all ½ point. The student lost three ½ points for the `isFriend` call, the `isFoe` call, and the counter update. One point was lost for not correctly identifying when to play dead, but since the student has some vague idea of what to do when the critter might be a foe, the response earned the ½ point for incrementing `numStepsDead` but lost the three ½ points for setting the color to black or orange and resetting `numStepsDead` to 0. The student earned a total of 1½ points for part (a).

In part (b) the student correctly identifies one case (dead) and correctly returns `null`. The student earned 1 point for part (b).

The total score of 2½ points was rounded up to 3.