# AP® COMPUTER SCIENCE A
# 2010 GENERAL SCORING GUIDELINES

**Apply the question-specific rubric first.** To maintain scoring intent, a single error is generally accounted for only once per question thereby mitigating multiple penalties for the same error. The error categorization below is for cases not adequately covered by the question-specific rubric. Note that points can only be deducted if the error occurs in a part that has earned credit via the question-specific rubric. Any particular error is **penalized only once** in a question, even if it occurs on different parts of that question.

## Nonpenalized Errors

spelling/case discrepancies if no ambiguity*

local variable not declared if others are declared in some part

use keyword as identifier

`[]` vs. `()` vs. `<>`

= 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; *e.g., precondition check*

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

missing { } where indentation clearly conveys intent and { } used elsewhere

default constructor called without parens; *e.g.,* `new Fish;`

missing `( )` on parameterless method call

missing `( )` around `if/while` conditions

missing `;` when majority are present

missing `public` on class or constructor header

extraneous `[]` when referencing entire array

extraneous size in array declaration, *e.g.,* `int[size] nums = new int[size];`

## Minor Errors (1/2 point)

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

local variables used but none declared

missing `new` in constructor call

modifying a constant (`final`)

use `equals` or `compareTo` method on primitives, *e.g.,* `int x; …x.equals(val)`

array/collection access confusion (`[] get`)

assignment dyslexia, *e.g.,* `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`

*Applying **Minor Errors** (½ point)*: A minor error that occurs **exactly once** when the same concept is **correct two or more times** is regarded as an oversight and **not penalized**. A minor error **must be penalized** if it is the **only instance, one of two**, or occurs **two or more times**.

## Major Errors (1 point)

extraneous code that causes side effect; *e.g., information written to output*

interface or class name instead of variable identifier; *e.g.,* `Bug.move()` *instead of* `aBug.move()`

`aMethod(obj)` instead of `obj.aMethod()`

attempt to use private data or method when not accessible

destruction of persistent data *(e.g., changing value referenced by parameter)*

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

`void` method (or constructor) returns a value

** Spelling and case discrepancies for identifiers fall under the "nonpenalized" category only if the correction can be **unambiguously** inferred from context; for example, "`ArrayList`" instead of "`ArrayList`". As a counter example, note that if a student declares "`Bug bug;`" then uses "`Bug.move()`" instead of "`bug.move()`", the context does **not** allow for the reader to assume the object instead of the class.*

# AP® COMPUTER SCIENCE A
# 2010 SCORING GUIDELINES

## Question 4: GridChecker (GridWorld)

**Part (a)** `actorWithMostNeighbors` **4 points**

***Intent:*** *Identify and return actor in* `this.gr` *with most neighbors; return* `null` *if no actors in grid*

**+1** Consider all occupied locations or all actors in grid
- **+1/2** Iterates over all occupied locations in `this.gr`
- **+1/2** Performs action using actor or location from `this.gr` within iteration

**+1 1/2** Determination of maximum number of neighbors
- **+1/2** Determines number of occupied neighboring locations* of a location
- **+1** Correctly determines maximum number of neighbors

**+1 1/2** Return actor
- **+1/2** Returns reference to `Actor` (not `Location`)
- **+1** Returns reference to a correct actor; `null` if no actors in `this.gr`

**Note: This may be done using* `getOccupiedAdjacentLocations`, `getNeighbors`, *or an iterative* `get` *of surrounding locations*

**Part (b)** `getOccupiedWithinTwo` **5 points**

***Intent:*** *Return list of all occupied locations within 2 rows/columns of parameter, parameter excluded*

**+1/2** Creates and initializes local variable to hold collection of locations

**+2** Consider surrounding locations
- **+1/2** Considers at least two locations 1 row and/or 1 column away from parameter
- **+1/2** Considers at least two locations 2 rows and/or 2 columns away from parameter
- **+1** Correctly identifies all and only valid locations within 2 rows and 2 columns of parameter

**+1** Collect occupied locations†
- **+1/2** Adds any location object to collection
- **+1/2** Adds location to collection only if occupied

**+1 1/2** Return list of locations
- **+1/2** Returns reference to a list of locations
- **+1/2** List contains all and only identified locations†
- **+1/2** Parameter `loc` excluded from returned list

†*Note: Duplication of locations in returned list is not penalized*

***Usage:*** **–½** parameter dyslexia in new `Location` constructor invocation

# AP® COMPUTER SCIENCE A
# 2010 CANONICAL SOLUTIONS

## Question 4: GridChecker (GridWorld)

**Part (a):**

```
public Actor actorWithMostNeighbors() {
  if (0 == this.gr.getOccupiedLocations().size()) {
    return null;
  }
  Location where = null;
  int most = -1;
  for (Location loc : this.gr.getOccupiedLocations()) {
    if (most < this.gr.getOccupiedAdjacentLocations(loc).size()) {
      most = this.gr.getOccupiedAdjacentLocations(loc).size();
      where = loc;
    }
  }
  return this.gr.get(where);
}

// Alternative solution (uses getNeighbors):

public Actor actorWithMostNeighbors() {
  if (0 == this.gr.getOccupiedLocations().size()) {
    return null;
  }
  Location where = this.gr.getOccupiedLocations().get(0);
  for (Location loc : this.gr.getOccupiedLocations()) {
    if (this.gr.getNeighbors(where).size() <
this.gr.getNeighbors(loc).size()) {
      where = loc;
    }
  }
  return this.gr.get(where);
}
```

These canonical solutions serve an expository role, depicting general approaches to a solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

# AP® COMPUTER SCIENCE A
# 2010 CANONICAL SOLUTIONS

## Question 4: GridChecker (GridWorld) (continued)

**Part (b):**

```
public List<Location> getOccupiedWithinTwo(Location loc) {
  List<Location> occupied = new ArrayList<Location>();
  for (int row = loc.getRow() - 2; row <= loc.getRow() + 2; row++) {
    for (int col = loc.getCol() - 2; col <= loc.getCol() + 2; col++) {
      Location loc1 = new Location(row, col);
      if (gr.isValid(loc1) && this.gr.get(loc1) != null &&
!loc1.equals(loc)) {
        occupied.add(loc1);
      }
    }
  }
  return occupied;
}

// Alternative solution (uses getOccupiedLocations):

public List<Location> getOccupiedWithinTwo(Location loc) {
  List<Location> occupied = new ArrayList<Location>();
  for (Location loc1 : this.gr.getOccupiedLocations()) {
    if ((Math.abs(loc.getRow() - loc1.getRow()) <= 2)
         && (Math.abs(loc.getCol() - loc1.getCol()) <= 2)
         && !loc1.equals(loc)) {
      occupied.add(loc1);
    }
  }
  return occupied;
}
```

These canonical solutions serve an expository role, depicting general approaches to a solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

4A.

Complete method `actorWithMostNeighbors` below.

```
/** @return an Actor in the grid gr with the most neighbors; null if no actors in the grid.
 */
public Actor actorWithMostNeighbors()
{
   ArrayList<Location> occupiedLocs = gr.getOccupiedLocations();
   if (occupiedLocs.size()==0)
     return (null);
   Location loc = OccupiedLocs.get(0)
   for(int i=0; i < OccupiedLocs; i++)
   {
      ArrayList<Location> temp =(gr.getOccapiedAdjacentLocations
                                  (occupiedLocs,get(i)))
      if(temp.size()>=(gr.getOccupiedAdjacentLocations(loc)).size())
      {
          loc = OccupiedLocs.get(i);
      }
   }
   return(gr.get(loc));
}
```

Part (b) begins on page 18.

4A.2

Complete method `getOccupiedWithinTwo` below.

```
/** Returns a list of all occupied locations in the grid gr that are within 2 rows
 *  and 2 columns of loc. The object references in the returned list may appear in any order.
 *  @param loc a valid location in the grid gr
 *  @return a list of all occupied locations in the grid gr that are within 2 rows
 *          and 2 columns of loc.
 */
public List<Location> getOccupiedWithinTwo(Location loc)
{
   List<Location> OccupiedLocs = new List<Location>();
   for(int r=loc.getRow()-2; r<=loc.getRow()+2; r++)
   {
      for(int c=loc.getCol()-2; c<=loc.getCol()+2; c++)
      {
         Location temp= new Location(r,c);
         if(gr.isValid(temp))
         {
            if((gr.get(temp) instanceof Actor) && (temp.compareTo(loc)!=0)
            {
               occupiedLocs.add(temp);
            }
         }
      }
   }
   return(occupiedLocs);
}
```

**GO ON TO THE NEXT PAGE.**

4B,

Complete method `actorWithMostNeighbors` below.

```
/** @return an Actor in the grid gr with the most neighbors; null if no actors in the grid.
 */
public Actor actorWithMostNeighbors()
{
   Actor winner = null;
   int most = 0;
   for (Location loc : gr.getOccupiedLocations())
   {

      Actor a = gr.get(loc);
      if ( gr.getNeighbors(loc).size() > most )
      {
        most = gr.getNeighbors(loc).size();
        winner = a;
      }
   }
   return winner;
}
```

Part (b) begins on page 18.

4B2

Complete method `getOccupiedWithinTwo` below.

```
/** Returns a list of all occupied locations in the grid gr that are within 2 rows
 *  and 2 columns of loc. The object references in the returned list may appear in any order.
 *  @param loc a valid location in the grid gr
 *  @return a list of all occupied locations in the grid gr that are within 2 rows
 *          and 2 columns of loc.
 */
public List<Location> getOccupiedWithinTwo(Location loc)
{
    List<Location> locs = new List<Location>;
    int myRow = loc.getRow();
    int myCol = loc.getCol();

    for (int i = -2; i <= 2; i++)
    {
        for (int k = -2; k <= 2; k++)
        {
            Location temp = new Location(myRow + i, myCol + i);
            if ( gr.isValid (temp))
            {
                locs.add ( gr.get (temp));
            }
        }
    return locs;
}
```

4C.

Complete method `actorWithMostNeighbors` below.

```
/** @return an Actor in the grid gr with the most neighbors; null if no actors in the grid.
 */
public Actor actorWithMostNeighbors()
    Actor a= new Actor;
    for (Actor i : gr){
        a = i;
        If (a.get().getEmptyAdjacentLocations() < i.get().getEmpty
                                                    AdjacentLocation())
        a = i }

    return a;
```

Part (b) begins on page 18.

$_4C_2$

Complete method `getOccupiedWithinTwo` below.

```
/** Returns a list of all occupied locations in the grid gr that are within 2 rows
 *  and 2 columns of loc. The object references in the returned list may appear in any order.
 *  @param loc a valid location in the grid gr
 *  @return a list of all occupied locations in the grid gr that are within 2 rows
 *          and 2 columns of loc.
 */
public List<Location> getOccupiedWithinTwo(Location loc)
```

~~return loc.getOccupiedAdjacentLocations();~~
~~getOccupiedAdjacentLocations();~~

```
ArrayList <Location> loc = new ArrayList <Location>();

for (int i=(loc.getRow()-2); int i =(loc.getRow()+2); i++){
 for (int b= (loc.getCol()-2); int b= (loc.getCol()+2); b++){
    If ( Actor[i][b]. isOccupied ){
       Loc.add (Actor. get());
    }
}

return loc;
```

**GO ON TO THE NEXT PAGE.**

# AP® COMPUTER SCIENCE A
# 2010 SCORING COMMENTARY

## Question 4

### Overview

This question involved reasoning about the code from the GridWorld case study, emphasizing use of the two-dimensional grid (rather than critter or actor definitions) and ArrayList processing. Students were asked to implement two unrelated methods of a `GridChecker` class. In part (a), `actorWithMostNeighbors`, students were required to retrieve all occupied locations in a grid, determine the number of neighbors of each, and return the actor with the most neighbors or null if the grid contained no actors. This could be accomplished by calling the `getOccupiedLocations` method of the grid object, then iterating over those locations, using each as the parameter to either the `getOccupiedAdjacentLocations` or `getNeighbors` method. Some students chose to iterate over the entire grid and check each location for the presence of an actor. Students needed to create, initialize and maintain a variable to track the maximum, and to return the correct actor reference. In part (b), `getOccupiedWithinTwo`, students were required to return a list containing all occupied locations within two rows and two columns of the given parameter. There were many viable solution approaches and the set of student solutions included most of them. Some students used the `getOccupiedLocations` method to access every occupied location in the grid, then needed to exclude those that were not within two rows and columns of the parameter. Some iterated over only locations that were within two rows and columns of the parameter, including only those in that range that were occupied. Others used the `getValidAdjacentLocations` method to access the locations within one row and one column of the parameter, then used those locations to access all occupied locations within two rows and columns of the parameter. In each case, students had to ensure that each location was valid and occupied and also had to exclude the parameter from the returned list.

### Sample: 4A
### Score: 8

In part (a) the student retrieves all occupied locations by calling `gr.getOccupiedLocations()` and performs an action on each location in the context of a loop but does not include `.size()` in the structure of the loop. The number of occupied neighboring locations is retrieved correctly using the size of the list returned by the method `gr.getOccupiedAdjacentLocations()`. The maximum number of neighbors is correctly determined by initializing `loc` to the first element of the occupied locations list, comparing the number of adjacent occupied locations of `loc` to the number of occupied adjacent locations of each location in the occupied locations list, and maintaining the maximum variable. The student returns the actor found in the maintained location after the completion of the loop. If no occupied locations exist in the grid, `null` is returned. Part (a) earned 3½ points.

In part (b) the student does not correctly instantiate the list `occupiedLocs`. The student iterates through all locations that are within two rows and two columns of the parameter. The student correctly checks if the location is both valid and occupied before adding the location to the list. The student also correctly prevents the parameter `loc` from being added to the list. The correct list is then returned. Part (b) earned 4½ points.

## Question 4 (continued)

**Sample: 4B**
**Score: 6**

In part (a) the student retrieves all occupied locations by calling `gr.getOccupiedLocations()` and performs an action on each location in the context of a loop. The number of occupied neighboring locations is retrieved correctly using the size of the list returned by the method `gr.getNeigbors()`. The maximum number of neighbors is correctly determined by initializing `most` to 0, comparing `most` to the size of each `gr.getNeighbors()` list, and maintaining the `most` variable. The student also maintains an actor variable when the number of neighbors is greater than `max` and returns that actor after the completion of the loop. However, when all actors in the grid have zero neighbors, the check will never evaluate to true. Since the `Actor` variable `winner` will never be updated in this case, the student did not earn the "Returns reference to a correct actor" point. Part (a) earned 3 points.

In part (b) the student does not correctly instantiate the list `locs`. The student iterates through all locations that are within two rows and two columns of the parameter and verifies that each location is valid. However, the student does not ensure that the location is occupied. The student adds actors instead of locations to the list `locs` and thus did not earn either of the "Collect occupied locations" ½ points. The student also does not attempt to exclude the parameter `loc` from the list. The final compiled list is then returned after the iteration. Part (b) earned 3 points.

**Sample: 4C**
**Score: 1.5 (rounded to 2)**

In part (a) the student correctly returns a reference to an actor. Part (a) earned ½ point.

In part (b) the student creates and initializes an ArrayList. The reference to the list is returned. Part (b) earned 1 point.