AP® Computer Science A
2004 Sample Student Responses

The materials included in these files are intended for noncommercial use by AP teachers for course and exam preparation; permission for any other use must be sought from the Advanced Placement Program®. Teachers may reproduce them, in whole or in part, in limited quantities, for face-to-face teaching purposes but may not mass distribute the materials, electronically or otherwise. This permission does not apply to any third-party copyrights contained herein. These materials and any copies made of them may not be resold, and the copyright notices must be retained as they appear here.

A3 A1

(a) Write the `PondStocker` method `numUnder`. Method `numUnder` returns the smallest number of fish that must be added to make the density of fish in the environment greater than `minDensity`. If the density of fish in the environment is already greater than `minDensity`, then `numUnder` returns zero. Recall that the `Environment` methods `numRows` and `numCols` return the number of rows and the number of columns, respectively, in an environment.

Complete method `numUnder` below.

```
// postcondition: returns the minimum number of fish that need to be
//                 added to make the population density greater than
//                 minDensity
private int numUnder()
{
  int numSpaces = theEnv.numRows()*theEnv.numCols();
  int filledSpaces = theEnv.numObjects();
  int needed = 0;

  while(((double)needed + filledSpaces) /(double)numSpaces) <= minDensity)
  {
    needed ++;
  }
  return needed;
}
```

Part (b) begins on page 14.

GO ON TO THE NEXT PAGE.

(b) Write the `PondStocker` method `randomLocation`. Method `randomLocation` returns a random location within the bounds of the environment.

In writing `randomLocation`, you may use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `randomLocation` below.

```
// postcondition: returns a random location within the bounds of theEnv
private Location randomLocation()
{
  Random randNumGen=RandomNumGenerator.getInstance();
  return newLocation(randNumGen.nextInt(theEnv.numRows()),
                     randNumGen.nextInt(theEnv.numCols()));
}
```

GO ON TO THE NEXT PAGE.

A_3

(c) Write the `PondStocker` method `addFish`. Method `addFish` adds `numToAdd Fish` to the environment at random locations that are not already occupied. You may use the two-parameter `Fish` constructor, so that the fish added have a random direction and color.

In writing `addFish`, you may call `randomLocation`. Assume that `randomLocation` works as specified, regardless of what you wrote in part (b). You may also use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `addFish` below.

```
// precondition:  0 <= numToAdd <= number of empty locations in theEnv
// postcondition: the number of fish in theEnv has been increased
//                by numToAdd; the fish added are placed at
//                random empty locations in theEnv
public void addFish(int numToAdd)
{
 for(int x=0; x<numToAdd; x++)
  {
   Location nextLocation=randomLocation();
   while(!theEnv.isEmpty(nextLocation))
    { nextLocation = randomLocation();
     }
    Fish current= new Fish(theEnv,nextLocation);
   }
  }
```

GO ON TO THE NEXT PAGE.

A3 B1

(a) Write the `PondStocker` method `numUnder`. Method `numUnder` returns the smallest number of fish that must be added to make the density of fish in the environment greater than `minDensity`. If the density of fish in the environment is already greater than `minDensity`, then `numUnder` returns zero. Recall that the `Environment` methods `numRows` and `numCols` return the number of rows and the number of columns, respectively, in an environment.

Complete method `numUnder` below.

```
// postcondition: returns the minimum number of fish that need to be
//                added to make the population density greater than
//                minDensity
private int numUnder() {
        Int base = theEnv.numRows() * theEnv.numCols() ;
        Locatible [ ] theFishes = theEnv.allObjects();
        int numFish= theFishes.length ;
        double currentDensity = (Double) numfish / base ;
        while ( currentDensity < minDensity){
             numFish++;
            currentDensity = (Double) numfish / base; }
         int numUnder= numFish - theFishes.length;
           return numUnder; }
```

Part (b) begins on page 14.

GO ON TO THE NEXT PAGE.

B_2

(b) Write the `PondStocker` method `randomLocation`. Method `randomLocation` returns a random location within the bounds of the environment.

In writing `randomLocation`, you may use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `randomLocation` below.

```
// postcondition: returns a random location within the bounds of theEnv
private Location randomLocation() {
        Random rand = New Random();
        int randRow = rand.nextInt (theEnv.numRows);
        int randCol = rand.nextInt (theEnv.numCols);
        Location randomLocation = New Location(radRow, randCol);
        If ( isValid (randomLocation) == true)
             return randomLocation; }
```

GO ON TO THE NEXT PAGE.

B_3

(c) Write the `PondStocker` method `addFish`. Method `addFish` adds `numToAdd Fish` to the environment at random locations that are not already occupied. You may use the two-parameter `Fish` constructor, so that the fish added have a random direction and color.

In writing `addFish`, you may call `randomLocation`. Assume that `randomLocation` works as specified, regardless of what you wrote in part (b). You may also use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `addFish` below.

```
// precondition:  0 <= numToAdd <= number of empty locations in theEnv
// postcondition: the number of fish in theEnv has been increased
//                by numToAdd; the fish added are placed at
//                random empty locations in theEnv
public void addFish(int numToAdd){

        for (int i=0; i ≤ numToAdd ; i++){
              Location randomLocation= new Location();
           if ( isEmpty (randomLocation)== true){
                 theEnv.add (new Fish(theEnv , randomLocation, theEnv.randomDirection),;
                     random Color(); } } }
```

GO ON TO THE NEXT PAGE.

A3 C1

(a) Write the `PondStocker` method `numUnder`. Method `numUnder` returns the smallest number of fish that must be added to make the density of fish in the environment greater than `minDensity`. If the density of fish in the environment is already greater than `minDensity`, then `numUnder` returns zero. Recall that the `Environment` methods `numRows` and `numCols` return the number of rows and the number of columns, respectively, in an environment.

Complete method `numUnder` below.

```
// postcondition: returns the minimum number of fish that need to be
//                added to make the population density greater than
//                minDensity
private int numUnder()
```

```
private int numUnder()
{
    int moreFish=0;
    int cols= Environment.numCols;
    int rows= Environment.numRows;
    int size= cols * rows;
    int neededDens= minDensity * size;
    if( Environment.allObjects() | <= minDensity)

        moreFish = neededDens - Environment.allObjects();
    return moreFish;
}
```

Part (b) begins on page 14.

GO ON TO THE NEXT PAGE.

C2

(b) Write the `PondStocker` method `randomLocation`. Method `randomLocation` returns a random location within the bounds of the environment.

In writing `randomLocation`, you may use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `randomLocation` below.

```
// postcondition: returns a random location within the bounds of theEnv
private Location randomLocation()
```

```
private Location randomLocation()
{
    int cols=Environment.numCols();
    int rows=Environment.numRows();
    x= Random.nextInt(rows+1);
    y= Random.nextInt(cols+1);
    return x, y;
}
```

GO ON TO THE NEXT PAGE.

C3

(c) Write the `PondStocker` method `addFish`. Method `addFish` adds `numToAdd` `Fish` to the environment at random locations that are not already occupied. You may use the two-parameter `Fish` constructor, so that the fish added have a random direction and color.

In writing `addFish`, you may call `randomLocation`. Assume that `randomLocation` works as specified, regardless of what you wrote in part (b). You may also use any of the accessible methods of the classes in the case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `addFish` below.

```
// precondition:  0 <= numToAdd <= number of empty locations in theEnv
// postcondition: the number of fish in theEnv has been increased
//                by numToAdd; the fish added are placed at
//                random empty locations in theEnv
public void addFish(int numToAdd)
```

```
public void addFish(int numToAdd)
{  ~~for(x=0; x<~~
   for(x=0; x<numToAdd; x++)
    {
      (Environment.add(Fish)).randomLocation();
    }
}
```

GO ON TO THE NEXT PAGE.