AP® Computer Science A
2005 Sample Student Responses

The College Board: Connecting Students to College Success

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 4,700 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three and a half million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities, and concerns.

Visit the College Board on the Web: www.collegeboard.com.
AP Central is the official online home for the AP Program and Pre-AP: apcentral.collegeboard.com.

1A A1

(a) Write the `Hotel` method `requestRoom`. Method `requestRoom` attempts to reserve a room in the hotel for a given guest. If there are any empty rooms in the hotel, one of them will be assigned to the named guest and the newly created reservation is returned. If there are no empty rooms, the guest is added to the end of the waiting list and `null` is returned.

Complete method `requestRoom` below.

```
// if there are any empty rooms (rooms with no reservation),
// then create a reservation for an empty room for the
// specified guest and return the new Reservation;
// otherwise, add the guest to the end of waitList
// and return null
public Reservation requestRoom(String guestName)
{
  for (int k = 0; k< rooms.length; k++)
  {
    if(rooms[k] = null)
    {
      rooms[k] = new Reservation (guestName, k);
      return rooms[k];
    }
  }

  waitList.add(guestName);
  return null;
}
```

Part (b) begins on page 6.

1A A2

(b) Write the `Hotel` method `cancelAndReassign`. Method `cancelAndReassign` releases a previous reservation. If the waiting list for the hotel contains any names, the vacated room is reassigned to the first person at the beginning of the list. That person is then removed from the waiting list and the newly created reservation is returned. If no one is waiting, the room is marked as empty and `null` is returned.

In writing `cancelAndReassign` you may call any accessible methods in the `Reservation` and `Hotel` classes. Assume that these methods work as specified.

Complete method `cancelAndReassign` below.

```
// release the room associated with parameter res, effectively
// canceling the reservation;
// if any names are stored in waitList, remove the first name
// and create a Reservation for this person in the room
// reserved by res; return that new Reservation;
// if waitList is empty, mark the room specified by res as empty and
// return null
// precondition:  res is a valid Reservation for some room
//                in this hotel
public Reservation cancelAndReassign(Reservation res)
{
  rooms[res.getRoomNumber()] = null;

  if(waitList.size() > 0)
  {
    return requestRoom(waitList.remove(0));
  }
  else
    return null;
}
```

GO ON TO THE NEXT PAGE.

1A B1

(a) Write the `Hotel` method `requestRoom`. Method `requestRoom` attempts to reserve a room in the hotel for a given guest. If there are any empty rooms in the hotel, one of them will be assigned to the named guest and the newly created reservation is returned. If there are no empty rooms, the guest is added to the end of the waiting list and `null` is returned.

Complete method `requestRoom` below.

```
// if there are any empty rooms (rooms with no reservation),
// then create a reservation for an empty room for the
// specified guest and return the new Reservation;
// otherwise, add the guest to the end of waitList
// and return null
public Reservation requestRoom(String guestName){

    for(int x = 0 ; x < rooms.length ; x++ ) {

        if( rooms [x] == null ) {

return  new Reservation ( guestName, rooms [x]. getRoomNumber() )

    }

else

    {

      waitList.add ( guestName );

      return  null;

    }
```

Part (b) begins on page 6.

1 A B2

(b) Write the `Hotel` method `cancelAndReassign`. Method `cancelAndReassign` releases a previous reservation. If the waiting list for the hotel contains any names, the vacated room is reassigned to the first person at the beginning of the list. That person is then removed from the waiting list and the newly created reservation is returned. If no one is waiting, the room is marked as empty and `null` is returned.

In writing `cancelAndReassign` you may call any accessible methods in the `Reservation` and `Hotel` classes. Assume that these methods work as specified.

Complete method `cancelAndReassign` below.

```
// release the room associated with parameter res, effectively
// canceling the reservation;
// if any names are stored in waitList, remove the first name
// and create a Reservation for this person in the room
// reserved by res; return that new Reservation;
// if waitList is empty, mark the room specified by res as empty and
// return null
// precondition:  res is a valid Reservation for some room
//                in this hotel
public Reservation cancelAndReassign(Reservation res) {
        int emptyRoom = 0;
        for ( int x = 0 ; x < rooms.length ; x++ ) {

            if( res.getRoomNumber() == rooms[x].getRoomNumber() ) {

                rooms[x] = null;
                emptyRoom = rooms[x].getRoomNumber();
            }

        }

    if ( waitList.size() != 0 ) {

        return new Reservation ( (String) waitList.get(0), emptyRoom );

            }
    else
        {
        return null;
        }

}
```

GO ON TO THE NEXT PAGE.

1A C,

(a) Write the `Hotel` method `requestRoom`. Method `requestRoom` attempts to reserve a room in the hotel for a given guest. If there are any empty rooms in the hotel, one of them will be assigned to the named guest and the newly created reservation is returned. If there are no empty rooms, the guest is added to the end of the waiting list and `null` is returned.

Complete method `requestRoom` below.

```
// if there are any empty rooms (rooms with no reservation),
// then create a reservation for an empty room for the
// specified guest and return the new Reservation;
// otherwise, add the guest to the end of waitList
// and return null
public Reservation requestRoom(String guestName)
{
  boolean hasRoom = false;
  for(int x = 0; (x < rooms.length()) && (hasRoom == false); x++)
  {
    if(rooms[x] == null)
    {
      rooms[x] = new Reservation(guestName, rooms[x].getRoomNumber());
      hasRoom = true;
    }
  }
  waitList.add(guestName)
}
```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

1A C2

(b) Write the `Hotel` method `cancelAndReassign`. Method `cancelAndReassign` releases a previous reservation. If the waiting list for the hotel contains any names, the vacated room is reassigned to the first person at the beginning of the list. That person is then removed from the waiting list and the newly created reservation is returned. If no one is waiting, the room is marked as empty and `null` is returned.

In writing `cancelAndReassign` you may call any accessible methods in the `Reservation` and `Hotel` classes. Assume that these methods work as specified.

Complete method `cancelAndReassign` below.

```
// release the room associated with parameter res, effectively
// canceling the reservation;
// if any names are stored in waitList, remove the first name
// and create a Reservation for this person in the room
// reserved by res; return that new Reservation;
// if waitList is empty, mark the room specified by res as empty and
// return null
// precondition: res is a valid Reservation for some room
//               in this hotel
public Reservation cancelAndReassign(Reservation res)
{
  res = new Reservation(res.getRoomNumber(), waitList.get(0));
  waitList.remove(0)
}
```

(a) Write the complete class declaration for the class `Advance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. Tickets purchased ten or more days in advance cost $30; tickets purchased nine or fewer days in advance cost $40.

```
public class Advance extends Ticket()
{
    private double price;
    public Advance(int daysInAdvance)
    {
        if(daysInAdvance > 9)
            price = 30;
        else
            price = 40;
    }
    public double getPrice()
    {
        return price;
    }
}
```

(b) Write the complete class declaration for the class `StudentAdvance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. The `toString` method should include a notation that a student ID is required for this ticket. A `StudentAdvance` ticket costs half of what that `Advance` ticket would normally cost. If the pricing scheme for `Advance` tickets changes, the `StudentAdvance` price should continue to be computed correctly with no code modifications to the `StudentAdvance` class.

```
public class StudentAdvance extends Advance()
{
    private double price;
    public StudentAdvance (int daysInAdvance)
    {
        super(daysInAdvance);
        price = (super.getPrice)/2;
    }
    public String toString()
    {
        return super.toString + " (Student ID required)";
    }
```

(a) Write the complete class declaration for the class `Advance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. Tickets purchased ten or more days in advance cost $30; tickets purchased nine or fewer days in advance cost $40.

```
public Advance extends Ticket
{

  public Advance(int theDaysInAdvance)
  {daysInAdvance = theDaysInAdvance;
  {

  public double getPrice()
  {
   {if(daysInAdvance >= 10)
     double cost = 30.00;
   else
     cost = 40.00;
    {
   return cost;
   {

  public int daysInAdvance;
  public double cost;

}
```

(b) Write the complete class declaration for the class `StudentAdvance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. The `toString` method should include a notation that a student ID is required for this ticket. A `StudentAdvance` ticket costs half of what that `Advance` ticket would normally cost. If the pricing scheme for `Advance` tickets changes, the `StudentAdvance` price should continue to be computed correctly with no code modifications to the `StudentAdvance` class.

```
public StudentAdvance extends Advance
{
  public StudentAdvance(int theDaysInAdvance)
  {super (daysInAdvance)
  {
public String toString()
  {
   return "Number:" +super.serialNumber + "\nPrice:" +
          getPrice() + "Student ID required";
  {
public double getPrice()
  {
double StudentCost = super.getPrice() / 2;
    return StudentCost;
  {
public double studentCost;
```

2A C.

(a) Write the complete class declaration for the class `Advance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. Tickets purchased ten or more days in advance cost $30; tickets purchased nine or fewer days in advance cost $40.

```
public class Advance extends Ticket
{
    int numOfDays=0;
    double price = 0.0;

    public Advance(int numDays)
    {
        numOfDays= numDays;
    }

    public double ticketPrice( )
    {
      if(numOfDays >= 10)
         price = 30.0;
       else if(numOfDays <=9)
         price= 40.0;
    }

    public String toString( )
    {
      return "Number:" + serialNumber +
                  "\nPrice:" + price;
    }
```

GO ON TO THE NEXT PAGE.

2A C2

(b) Write the complete class declaration for the class `StudentAdvance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. The `toString` method should include a notation that a student ID is required for this ticket. A `StudentAdvance` ticket costs half of what that `Advance` ticket would normally cost. If the pricing scheme for `Advance` tickets changes, the `StudentAdvance` price should continue to be computed correctly with no code modifications to the `StudentAdvance` class.

```
public class StudentAdvance extends Advance
{

   public StudentAdvance(int numDays)
   {
     numOfDays = numDays;
   }
  public double ticketPrice
  {  if(numOfDays >= 10)
        price = 15.0;
     else if(numOfDays <= 9)
        price = 20.0;
  }

  public String toString()
  {
     return "Number;" + serialNumber +
            "\nPrice;" + price + "Student" +
            "ID required";
  }

}
```

GO ON TO THE NEXT PAGE.

(a) Override the `nextLocation` method for the `ZigZagFish` class. The `nextLocation` method returns the cell diagonally forward to the right, the cell diagonally forward to the left, or the cell that the `ZigZagFish` currently occupies, according to the description given at the beginning of this question.

In writing `nextLocation`, you may use any of the accessible methods of the classes in the case study.

Complete method `nextLocation` below.

```
// returns the forward diagonal cell to the left or right of this fish
// (depending on willZigRight) if that cell is empty;
// otherwise, returns this fish's current location
// postcondition: the state of this ZigZagFish is unchanged
protected Location nextLocation()
{
    Environment env = environment();
    Location oneInFront = env.getNeighbor(location(), direction());
    if (willZigRight)
    {
        Location zig = env.getNeighbor(oneInFront,
                                       direction.toRight());
    }
    else
    {
        Location zig = env.getNeighbor(oneInFront,
                                       direction.toLeft());
    }
    if (env.isEmpty(zig))
        return zig;
    else
        return location();
}
```

(b) Override the `move` method for the `ZigZagFish` class. This method should change the location and direction of the fish as needed, according to the rules of movement described at the beginning of the question. In addition, the state of the fish must be updated.

In writing `move`, you may call `nextLocation`. Assume that `nextLocation` works as specified, regardless of what you wrote in part (a). You may also use any of the accessible methods of the classes in the case study.

Complete method `move` below.

```
// moves this ZigZagFish diagonally (as specified in nextLocation) if
// possible; otherwise, reverses direction without moving;
// after a diagonal move, willZigRight is updated
protected void move()
{
    Location nextLoc = nextLocation();
    if(!nextLoc.equals(location()))
    {
        changeLocation(nextLoc);
        willZigRight = !willZigRight;
    }
    else
    {
        changeDirection(direction.reverse());
    }
}
```

3A B1

(a) Override the `nextLocation` method for the `ZigZagFish` class. The `nextLocation` method returns the cell diagonally forward to the right, the cell diagonally forward to the left, or the cell that the `ZigZagFish` currently occupies, according to the description given at the beginning of this question.

In writing `nextLocation`, you may use any of the accessible methods of the classes in the case study.

Complete method `nextLocation` below.

```
// returns the forward diagonal cell to the left or right of this fish
// (depending on willZigRight) if that cell is empty;
// otherwise, returns this fish's current location
// postcondition: the state of this ZigZagFish is unchanged
protected Location nextLocation()
{  Location newLoc;
   Location locInFront = environment().getNeighbor(location(), direction());

   if (willZigRight)
     newLoc = environment().getNeighbor(locInFront, direction().toRight());
   else
   newLoc = environment().getNeighbor(locInFront, direction().toLeft());

  return newLoc;
}
```

GO ON TO THE NEXT PAGE.

(b) Override the `move` method for the `ZigZagFish` class. This method should change the location and direction of the fish as needed, according to the rules of movement described at the beginning of the question. In addition, the state of the fish must be updated.

In writing `move`, you may call `nextLocation`. Assume that `nextLocation` works as specified, regardless of what you wrote in part (a). You may also use any of the accessible methods of the classes in the case study.

Complete method `move` below.

```
// moves this ZigZagFish diagonally (as specified in nextLocation) if
// possible; otherwise, reverses direction without moving;
// after a diagonal move, willZigRight is updated
protected void move()
{
    boolean update = willZigRight;
    Location nextLoc = nextLocation();
    Location locFront = environment().getNeighbor(nextLoc, direction());
    if (!locFront.isInEnv || !environment().isEmpty(locFront))
    {
        Direction oppDir = direction().reverse();
        changeDirection(oppDir);
    }
    changeLocation(nextLoc);
    if (update)
        willZigRight = false;
}
```

GO ON TO THE NEXT PAGE.

3A C,

(a) Override the `nextLocation` method for the `ZigZagFish` class. The `nextLocation` method returns the cell diagonally forward to the right, the cell diagonally forward to the left, or the cell that the `ZigZagFish` currently occupies, according to the description given at the beginning of this question.

In writing `nextLocation`, you may use any of the accessible methods of the classes in the case study.

Complete method `nextLocation` below.

```
// returns the forward diagonal cell to the left or right of this fish
// (depending on willZigRight) if that cell is empty;
// otherwise, returns this fish's current location
// postcondition: the state of this ZigZagFish is unchanged
protected Location nextLocation()
```

```
protected Location nextLocation()
{int zigzagFish1;
Direction zigzagFish1 = direction().forward;
Direction zigzagFish1 = new direction().right;
Direction zigzagFish1 = new direction().forward;
Direction zigzagFish1 = new direction().left;
if (emptyNbrs == 0)
  return location;
}
```

GO ON TO THE NEXT PAGE.

3A C2

(b) Override the `move` method for the `ZigZagFish` class. This method should change the location and direction of the fish as needed, according to the rules of movement described at the beginning of the question. In addition, the state of the fish must be updated.

In writing `move`, you may call `nextLocation`. Assume that `nextLocation` works as specified, regardless of what you wrote in part (a). You may also use any of the accessible methods of the classes in the case study.

Complete method `move` below.

```
// moves this ZigZagFish diagonally (as specified in nextLocation) if
// possible; otherwise, reverses direction without moving;
// after a diagonal move, willZigRight is updated
protected void move()
```

```
protected void move()
{
    debug.print("ZigZagFish " + toString()
                + " attempting to move.");

    Location nextLoc = nextLocation();

    if ( nexLocation.equals(location)
      {
          changeLocattion(nextLocation);

      }
    else
      changDirection(direction().reverse());

}
```

GO ON TO THE NEXT PAGE.

4A A1

(a) Write the `StudentRecord` method `average`. This method returns the average of the values in `scores` given a starting and an ending index.

Complete method `average` below.

```
// returns the average (arithmetic mean) of the values in scores
// whose subscripts are between first and last, inclusive
// precondition:  0 <= first <= last < scores.length
private double average(int first, int last)
```

```
private double average(int first, int last)
{ double num = last - first + 1;
  double sum = 0;
  for (int k = first; k <= last; k++)
     sum += scores[k];
  return sum/num;
}
```

(b) Write the `StudentRecord` method `hasImproved`.

Complete method `hasImproved` below.

```
// returns true if each successive value in scores is greater
// than or equal to the previous value;
// otherwise, returns false
private boolean hasImproved()
```

```
private boolean hasImpared()
{ boolean impared=true;
  for (int k=0; k<scores.length-1; k++)
     if (scores[k]>scores[k+1])
        impared=false;
  return impared;
}
```

Part (c) begins on page 20.

(c) Write the `StudentRecord` method `finalAverage`.

In writing `finalAverage`, you must call the methods defined in parts (a) and (b). Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b).

Complete method `finalAverage` below.

```
// if the values in scores have improved, returns the average
// of the elements in scores with indexes greater than or equal
// to scores.length/2;
// otherwise, returns the average of all of the values in scores
public double finalAverage()
```

```
public double finalAverage()
{
  if (hasImproved)
    return average(scores.length/2, scores.length-1)
  else
    return average(0, scores.length-1)
}
```

4A B1

(a) Write the `StudentRecord` method `average`. This method returns the average of the values in `scores` given a starting and an ending index.

Complete method `average` below.

```
// returns the average (arithmetic mean) of the values in scores
// whose subscripts are between first and last, inclusive
// precondition:  0 <= first <= last < scores.length
private double average(int first, int last)
{
    double scoreSize = (last - first) (double);
    int totalScore = 0;
    for(int i = first; i <= last; i++)
    {
        totalScore += scores[i];
    }

    double averageScore = totalScore / scoreSize;
    return averageScore;
}
```

GO ON TO THE NEXT PAGE.

4A B2

(b) Write the `StudentRecord` method `hasImproved`.

Complete method `hasImproved` below.

```
// returns true if each successive value in scores is greater
// than or equal to the previous value;
// otherwise, returns false
private boolean hasImproved()
{
    int i = 0;
    boolean showImprove = true;
    while ( showImprove && i < scores.length-1 )
    {
        IF ( scores[i+1] >= scores[i] )
            showImprove = true;
        else
            showImprove = false;
        i++;
    }
    return showImprove;
}
```

Part (c) begins on page 20.

4A B3

(c) Write the `StudentRecord` method `finalAverage`.

In writing `finalAverage`, you must call the methods defined in parts (a) and (b). Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b).

Complete method `finalAverage` below.

```
// if the values in scores have improved, returns the average
// of the elements in scores with indexes greater than or equal
// to scores.length/2;
// otherwise, returns the average of all of the values in scores
public double finalAverage()
{
    double finAverage;
    if( hasImproved() )
    {
        int sum = 0;
        int j = scores.length/2;
        while( j < scores.length )
        {
            sum += scores[j];
            j++;
        }
        finAverage = (double) sum / (scores.length/2);
    }
    else
        finAverage = average(0, scores.length);
    return finAverage;
}
```

GO ON TO THE NEXT PAGE.

4A C1

(a) Write the `StudentRecord` method `average`. This method returns the average of the values in `scores` given a starting and an ending index.

Complete method `average` below.

```
// returns the average (arithmetic mean) of the values in scores
// whose subscripts are between first and last, inclusive
// precondition:  0 <= first <= last < scores.length
private double average(int first, int last)
{
  double b;

  if (first > last)
    b = (first + last) / scores[].length;
  else
    b = last /(scores[].length/2);



  return b;
}
```

4A C2

(b) Write the `StudentRecord` method `hasImproved`.

Complete method `hasImproved` below.

```
// returns true if each successive value in scores is greater
// than or equal to the previous value;
// otherwise, returns false
private boolean hasImproved()
{
  boolean c;
   for ( int k=0; k <= scores[].length; k ++)
   {  if ( scores[k] < scores[k+1] )
       c = true;
     else
       c = false;
    }
   return c;
}
```

Part (c) begins on page 20.

4A C3

(c) Write the `StudentRecord` method `finalAverage`.

In writing `finalAverage`, you must call the methods defined in parts (a) and (b). Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b).

Complete method `finalAverage` below.

```
// if the values in scores have improved, returns the average
// of the elements in scores with indexes greater than or equal
// to scores.length/2;
// otherwise, returns the average of all of the values in scores
public double finalAverage()
{
  if ( hasImproved == true)
    return average ( scores, length/2, scores.length);
  else
    return average ( scores [0+ scores.length], scores. length);
}
```

GO ON TO THE NEXT PAGE.