AP® Computer Science A
2006 Free-Response Questions

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 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven 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: apcentral.collegeboard.com.

COMPUTER SCIENCE A
SECTION II
Time—1 hour and 45 minutes
Number of questions—4
Percent of total grade—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN Java.

Notes:
- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

GO ON TO THE NEXT PAGE.

1. An appointment scheduling system is represented by the following three classes: `TimeInterval`, `Appointment`, and `DailySchedule`. In this question, you will implement one method in the `Appointment` class and two methods in the `DailySchedule` class.

   A `TimeInterval` object represents a period of time. The `TimeInterval` class provides a method to determine if another time interval overlaps with the time interval represented by the current `TimeInterval` object. An `Appointment` object contains a time interval for the appointment and a method that determines if there is a time conflict between the current appointment and another appointment. The declarations of the `TimeInterval` and `Appointment` classes are shown below.

```
public class TimeInterval
{
  // returns true if interval overlaps with this TimeInterval;
  // otherwise, returns false
  public boolean overlapsWith(TimeInterval interval)
  {  /* implementation not shown */  }

  // There may be fields, constructors, and methods that are not shown.
}


public class Appointment
{
  // returns the time interval of this Appointment
  public TimeInterval getTime()
  {  /* implementation not shown */  }

  // returns true if the time interval of this Appointment
  // overlaps with the time interval of other;
  // otherwise, returns false
  public boolean conflictsWith(Appointment other)
  {  /* to be implemented in part (a) */  }

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

   (a) Write the `Appointment` method `conflictsWith`. If the time interval of the current appointment overlaps with the time interval of the appointment `other`, method `conflictsWith` should return `true`, otherwise, it should return `false`.

   Complete method `conflictsWith` below.

```
  // returns true if the time interval of this Appointment
  // overlaps with the time interval of other;
  // otherwise, returns false
  public boolean conflictsWith(Appointment other)
```

(b) A `DailySchedule` object contains a list of nonoverlapping `Appointment` objects. The `DailySchedule` class contains methods to clear all appointments that conflict with a given appointment and to add an appointment to the schedule.

```
public class DailySchedule
{
  // contains Appointment objects, no two Appointments overlap
  private ArrayList apptList;


  public DailySchedule()
  {  apptList = new ArrayList();  }


  // removes all appointments that overlap the given Appointment
  // postcondition: all appointments that have a time conflict with
  //                appt have been removed from this DailySchedule
  public void clearConflicts(Appointment appt)
  {  /* to be implemented in part (b) */  }


  // if emergency is true, clears any overlapping appointments and adds
  // appt to this DailySchedule; otherwise, if there are no conflicting
  // appointments, adds appt to this DailySchedule;
  // returns true if the appointment was added;
  // otherwise, returns false
  public boolean addAppt(Appointment appt, boolean emergency)
  {  /* to be implemented in part (c) */  }


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

Write the `DailySchedule` method `clearConflicts`. Method `clearConflicts` removes all appointments that conflict with the given appointment.

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

Complete method `clearConflicts` below.

```
// removes all appointments that overlap the given Appointment
// postcondition: all appointments that have a time conflict with
//                appt have been removed from this DailySchedule
public void clearConflicts(Appointment appt)
```

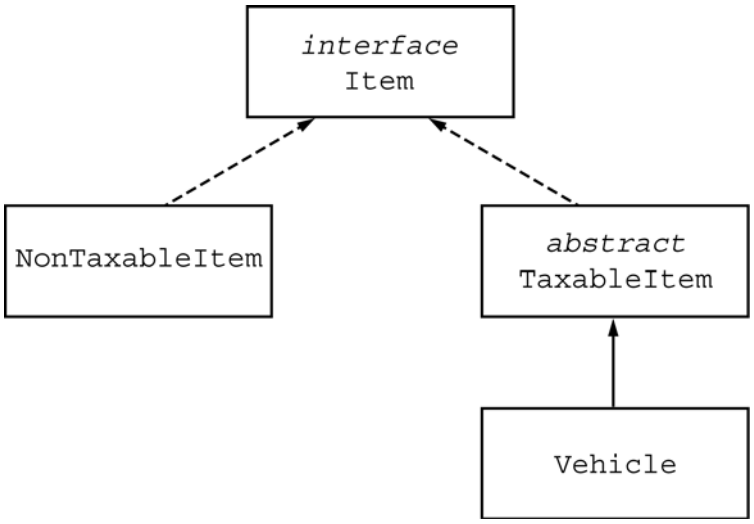
(c) Write the `DailySchedule` method `addAppt`. The parameters to method `addAppt` are an appointment and a `boolean` value that indicates whether the appointment to be added is an emergency. If the appointment is an emergency, the schedule is cleared of all appointments that have a time conflict with the given appointment and the appointment is added to the schedule. If the appointment is not an emergency, the schedule is checked for any conflicting appointments. If there are no conflicting appointments, the given appointment is added to the schedule. Method `addAppt` returns `true` if the appointment was added to the schedule; otherwise, it returns `false`.

In writing method `addAppt`, you may assume that `conflictsWith` and `clearConflicts` work as specified, regardless of what you wrote in parts (a) and (b).

Complete method `addAppt` below.

```
// if emergency is true, clears any overlapping appointments and adds
// appt to this DailySchedule; otherwise, if there are no conflicting
// appointments, adds appt to this DailySchedule;
// returns true if the appointment was added;
// otherwise, returns false
public boolean addAppt(Appointment appt, boolean emergency)
```

2. A set of classes is used to represent various items that are available for purchase. Items are either taxable or nontaxable. The purchase price of a taxable item is computed from its list price and its tax rate. The purchase price of a nontaxable item is simply its list price. Part of the class hierarchy is shown in the diagram below.



The definitions of the `Item` interface and the `TaxableItem` class are shown below.

```
public interface Item
{
  double purchasePrice();
}


public abstract class TaxableItem implements Item
{
  private double taxRate;


  public abstract double getListPrice();


  public TaxableItem(double rate)
  {  taxRate = rate;  }


  // returns the price of the item including the tax
  public double purchasePrice()
  {  /* to be implemented in part (a) */  }

}
```

(a) Write the `TaxableItem` method `purchasePrice`. The purchase price of a `TaxableItem` is its list price plus the tax on the item. The tax is computed by multiplying the list price by the tax rate. For example, if the tax rate is 0.10 (representing 10%), the purchase price of an item with a list price of $6.50 would be $7.15.

Complete method `purchasePrice` below.

```
// returns the price of the item including the tax
public double purchasePrice()
```

(b) Create the `Vehicle` class, which extends the `TaxableItem` class. A vehicle has two parts to its list price: a dealer cost and dealer markup. The list price of a vehicle is the sum of the dealer cost and the dealer markup.

For example, if a vehicle has a dealer cost of $20,000.00, a dealer markup of $2,500.00, and a tax rate of 0.10, then the list price of the vehicle would be $22,500.00 and the purchase price (including tax) would be $24,750.00. If the dealer markup were changed to $1,000.00, then the list price of the vehicle would be $21,000.00 and the purchase price would be $23,100.00.

Your class should have a constructor that takes dealer cost, the dealer markup, and the tax rate as parameters. Provide any private instance variables needed and implement all necessary methods. Also provide a public method `changeMarkup`, which changes the dealer markup to the value of its parameter.

3. Consider the following incomplete class that stores information about a customer, which includes a name and unique ID (a positive integer). To facilitate sorting, customers are ordered alphabetically by name. If two or more customers have the same name, they are further ordered by ID number. A particular customer is "greater than" another customer if that particular customer appears later in the ordering than the other customer.

```
public class Customer
{
  // constructs a Customer with given name and ID number
  public Customer(String name, int idNum)
  {  /* implementation not shown */  }

  // returns the customer's name
  public String getName()
  {  /* implementation not shown */  }

  // returns the customer's id
  public int getID()
  {  /* implementation not shown */  }


  // returns 0 when this customer is equal to other;
  //   a positive integer when this customer is greater than other;
  //   a negative integer when this customer is less than other
  public int compareCustomer(Customer other)
  {  /* to be implemented in part (a) */  }

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

(a) Write the `Customer` method `compareCustomer`, which compares this customer to a given customer, `other`. Customers are ordered alphabetically by name, using the `compareTo` method of the `String` class. If the names of the two customers are the same, then the customers are ordered by ID number. Method `compareCustomer` should return a positive integer if this customer is greater than `other`, a negative integer if this customer is less than `other`, and 0 if they are the same.

For example, suppose we have the following `Customer` objects.

```
Customer c1 = new Customer("Smith", 1001);
Customer c2 = new Customer("Anderson", 1002);
Customer c3 = new Customer("Smith", 1003);
```

The following table shows the result of several calls to `compareCustomer`.

Method Call	Result
`c1.compareCustomer(c1)`	0
`c1.compareCustomer(c2)`	a positive integer
`c1.compareCustomer(c3)`	a negative integer

Complete method `compareCustomer` below.

```
// returns 0 when this customer is equal to other;
//   a positive integer when this customer is greater than other;
//   a negative integer when this customer is less than other
public int compareCustomer(Customer other)
```

(b) A company maintains customer lists where each list is a sorted array of customers stored in ascending order by customer. A customer may appear in more than one list, but will not appear more than once in the same list.

Write method `prefixMerge`, which takes three array parameters. The first two arrays, `list1` and `list2`, represent existing customer lists. It is possible that some customers are in both arrays. The third array, `result`, has been instantiated to a length that is no longer than either of the other two arrays and initially contains `null` values. Method `prefixMerge` uses an algorithm similar to the merge step of a Mergesort to fill the array `result`. Customers are copied into `result` from the beginning of `list1` and `list2`, merging them in ascending order until all positions of `result` have been filled. Customers who appear in both `list1` and `list2` will appear at most once in `result`.

For example, assume that three arrays have been initialized as shown below.

`list1`

Arthur 4920	Burton 3911	Burton 4944	Franz 1692	Horton 9221	Jones 5554	Miller 9360	Nguyen 4339
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

`list2`

Aaron 1729	Baker 2921	Burton 3911	Dillard 6552	Jones 5554	Miller 9360	Noble 3335
[0]	[1]	[2]	[3]	[4]	[5]	[6]

`result`

null	null	null	null	null	null
[0]	[1]	[2]	[3]	[4]	[5]

In this example, the array `result` must contain the following values after the call `prefixMerge(list1, list2, result)`.

`result`

Aaron 1729	Arthur 4920	Baker 2921	Burton 3911	Burton 4944	Dillard 6552
[0]	[1]	[2]	[3]	[4]	[5]

In writing `prefixMerge`, you may assume that `compareCustomer` works as specified, regardless of what you wrote in part (a). Solutions that create any additional data structures holding multiple objects (e.g., arrays, `ArrayLists`, etc.) will not receive full credit.

Complete method `prefixMerge` below.

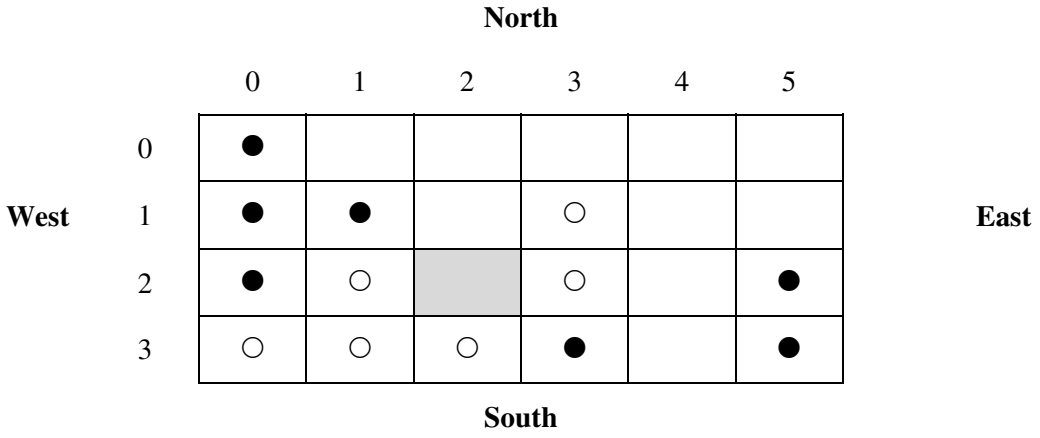
```
// fills result with customers merged from the
// beginning of list1 and list2;
// result contains no duplicates and is sorted in
// ascending order by customer
// precondition:  result.length > 0;
//                list1.length >= result.length;
//                list1 contains no duplicates;
//                list2.length >= result.length;
//                list2 contains no duplicates;
//                list1 and list2 are sorted in
//                ascending order by customer
// postcondition: list1, list2 are not modified
public static void prefixMerge(Customer[] list1,
                               Customer[] list2,
                               Customer[] result)
```

4. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

   Consider using the `BoundedEnv` class from the Marine Biology Simulation case study to model a game board. In this implementation of the `Environment` interface, each location has at most **four** neighbors. Those neighbors are determined by the `Environment` method `neighborsOf`.

   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 **three** 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	(0, 1)
2	(2, 2)
3	(0, 3)
4	(3, 4)
5	(1, 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 three neighboring locations — (2, 1), (3, 2), and (2, 3) — contain WHITE pieces. This move is the only available winning move on the above game board. Note that a BLACK piece dropped in column 1 would land in location (0, 1) and not result in a win because the neighboring location (0, 2) does not contain a BLACK piece.

The `Piece` class implements the `Locatable` interface and is defined as follows.

```
public class Piece implements Locatable
{
  // returns location of this Piece
  public Location location()
  {  /* implementation not shown */  }

  // returns color of this Piece
  public Color color()
  {  /* implementation not shown */  }

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

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

```
public class DropGame
{
  private Environment theEnv;  // contains Piece objects


  // returns null if no empty locations in column;
  // otherwise, returns the empty location with the
  // largest row index within the specified column;
  // precondition: 0 <= column < theEnv.numCols()
  public Location dropLocationForColumn(int column)
  {  /* to be implemented in part (a) */  }


  // returns true if dropping a piece of the given color into the
  // specified column matches color with three neighbors;
  // otherwise, returns false
  // precondition: 0 <= column < theEnv.numCols()
  public boolean dropMatchesNeighbors(int column, Color pieceColor)
  {  /* to be implemented in part (b) */  }


  // There may be fields, 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.

In writing `dropLocationForColumn`, you may use any methods defined in the `DropGame` class or accessible methods of the case study classes.

Complete method `dropLocationForColumn` below.

```
// returns null if no empty locations in column;
// otherwise, returns the empty location with the
// largest row index within the specified column;
// precondition: 0 <= column < theEnv.numCols()
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 three 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.

```
// returns true if dropping a piece of the given color into the
// specified column matches color with three neighbors;
// otherwise, returns false
// precondition: 0 <= column < theEnv.numCols()
public boolean dropMatchesNeighbors(int column, Color pieceColor)
```

END OF EXAM