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)
```

GO ON TO THE NEXT PAGE.

(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)
```

GO ON TO THE NEXT PAGE.

(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`.

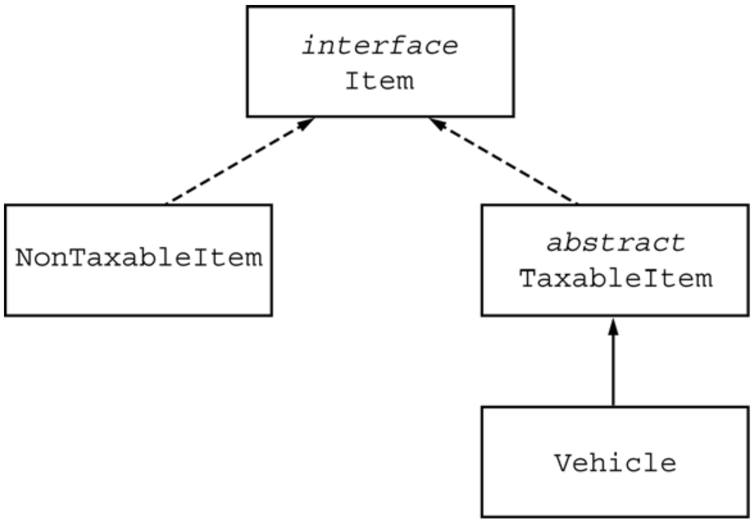
In writing method `addAppt`, you may assume that `conflictsWith` and `clearConflicts` work as specified, regardless of what you wrote in parts (a) and (b).

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)
```

GO ON TO THE NEXT PAGE.

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) */  }

}
```

GO ON TO THE NEXT PAGE.

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.
}
```

GO ON TO THE NEXT PAGE.

(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

Question 1: Daily Schedule

Part A:	`conflictsWith`	**1 1/2 points**

+1/2 call `OBJ1.overlapsWith(OBJ2)`
+1/2 access `getTime of other and this`
+1/2 return correct value

Part B:	`clearConflicts`	**3 points**

+2 loop over `apptList`
- **+1/2** reference `apptList` in loop body
- **+1/2** access appointment *in context of loop* (`apptList.get(i)`)
- **+1** access all appointments (cannot skip entries after a removal)

+1 remove conflicts *in context of loop*
- **+1/2** determine when conflict exists (must call `conflictsWith`)
- **+1/2** remove all conflicting appointments (and no others)

Part C:	`addAppt`	**4 1/2 points**

+1/2 test if emergency (*may limit to when emergency AND conflict exists*)
+1/2 clear conflicts if and only if emergency
(must not reimplement `clearConflicts` code)
+1/2 add `appt` if emergency

+2 non-emergency case
- **+1/2** loop over `apptList` (must reference `apptList` in body)
- **+1/2** access `apptList` element and check for `appt` conflicts *in context of loop*
- **+1/2** exit loop with state (conflict / no conflict) correctly determined
  (includes loop bound)
- **+1/2** add `appt` if and only if no conflict

+1 return true if any appointment added, false otherwise (must return both)

Usage: -1 if loop structure results in failure to handle empty `apptList`

AP® COMPUTER SCIENCE A
2006 CANONICAL SOLUTIONS

Question 1: Daily Schedule

PART A:

```
public boolean conflictsWith(Appointment other)
{
  return getTime().overlapsWith(other.getTime());
}
```

PART B:

```
public void clearConflicts(Appointment appt)
{
  int i = 0;
  while (i < apptList.size())
  {
    if (appt.conflictsWith((Appointment)(apptList.get(i))))
    {
      apptList.remove(i);
    }
    else
    {
      i++;
    }
  }
}
```

ALTERNATE SOLUTION

```
public void clearConflicts(Appointment appt)
{
  for (int i = apptList.size()-1; i >= 0; i--)
  {
    if (appt.conflictsWith((Appointment)apptList.get(i)))
    {
      apptList.remove(i);
    }
  }
}
```

Question 1: Daily Schedule (continued)

PART C:

```
public boolean addAppt(Appointment appt, boolean emergency)
{
  if (emergency)
  {
    clearConflicts(appt);
  }
  else
  {
    for (int i = 0; i < apptList.size(); i++)
    {
      if (appt.conflictsWith((Appointment)apptList.get(i)))
      {

        return false;
      }
    }
  }
  return apptList.add(appt);
```