# Question 1: Daily Schedule

## Part A: conflictsWith 1 1/2 points

1. +1/2 call `OBJ1.overlapsWith(OBJ2)`
2. +1/2 access `getTime` of other and this
3. +1/2 return correct value

## Part B: clearConflicts 3 points

1. +2 loop over `apptList`
2. +1/2 reference `apptList` in loop body
3. +1/2 access appointment in context of loop (``apptList.get(i)``)
4. +1 access all appointments (cannot skip entries after a removal)
5. +1 remove conflicts in context of loop
6. +1/2 determine when conflict exists (must call `conflictsWith`)
7. +1/2 remove all conflicting appointments (and no others)

## Part C: addAppt 4 1/2 points

1. +1/2 test if emergency (may limit to when emergency AND conflict exists)
2. +1/2 clear conflicts if and only if emergency
   - (must not reimplement `clearConflicts` code)
3. +1/2 add `appt` if emergency
4. +2 non-emergency case
   1. +1/2 loop over `apptList` (must reference `apptList` in body)
   2. +1/2 access `apptList` element and check for `appt` conflicts in context of loop
   3. +1/2 exit loop with state (conflict / no conflict) correctly determined
      - (includes loop bound)
   4. +1/2 add `appt` if and only if no conflict
5. +1 return true if any appointment added, false otherwise (must return both)

**Usage:** -1 if loop structure results in failure to handle empty `apptList`
**Question 2: Taxable Items (Design)**

<table>
<thead>
<tr>
<th>Part A: purchasePrice</th>
<th>2 1/2 points</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1 call getListPrice()</td>
<td></td>
</tr>
<tr>
<td>+1 calculate correct purchase price (<em>no penalty if truncate/round to 2 decimal places</em>)</td>
<td></td>
</tr>
<tr>
<td>+1/2 return calculated price</td>
<td></td>
</tr>
</tbody>
</table>

**Part B: Vehicle | 6 1/2 points**

+1/2 class Vehicle extends TaxableItem
+1/2 private double dealerCost
+1/2 private double dealerMarkup (*no penalty if also store tax in field*)

+2 1/2 constructor
  +1/2 Vehicle(double ?, double ?, double ?)  
    int/float (*OK if match fields*)
  +1 call parent constructor
  +1/2 attempt using super
  +1/2 correct call: super(rate) (*note: must be first line in method*)
  +1 initialize dealer cost and markup fields
  +1/2 attempt (must use parameters on RHS or in mutator call)
  +1/2 correct

+1 changeMarkup
+1/2 public void changeMarkup(double ?)  
  int/float (*OK if matches field; no penalty if returns reasonable value*)
  +1/2 assign parameter to markup field

+1 1/2 getListPrice
  +1 public double getListPrice()  
  +1/2 return sum of dealer cost and markup fields

**Note:** -1 usage if reimplement purchasePrice to do anything other than
return super.purchasePrice();
Question 3: Customer List

<table>
<thead>
<tr>
<th>Part A: compareCustomer</th>
<th>3 points</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1 1/2 perform comparison</td>
<td></td>
</tr>
<tr>
<td>+1/2 attempt (must call OBJ1.compareTo(OBJ2))</td>
<td></td>
</tr>
<tr>
<td>+1/2 correctly access and compare names</td>
<td></td>
</tr>
<tr>
<td>+1/2 correctly access and compare IDs</td>
<td></td>
</tr>
<tr>
<td>+1/2 return 0 if and only if this = other</td>
<td></td>
</tr>
<tr>
<td>+1/2 return positive if and only if this &gt; other</td>
<td></td>
</tr>
<tr>
<td>+1/2 return negative if and only if this &lt; other</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part B: prefixMerge</th>
<th>6 points</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1/2 initialize unique variables to index fronts of arrays</td>
<td></td>
</tr>
<tr>
<td>+1 1/2 loop over arrays to fill result</td>
<td></td>
</tr>
<tr>
<td>+1/2 attempt (must reference list1 and list2 inside loop)</td>
<td></td>
</tr>
<tr>
<td>+1 correct (lose this if add too few or too many Customer elements)</td>
<td></td>
</tr>
<tr>
<td>+1 1/2 compare array fronts (in context of loop)</td>
<td></td>
</tr>
<tr>
<td>+1/2 attempt (must call compareCustomer on array elements)</td>
<td></td>
</tr>
<tr>
<td>+1 correctly compare front Customer elements</td>
<td></td>
</tr>
<tr>
<td>+1 1/2 duplicate entries</td>
<td></td>
</tr>
<tr>
<td>+1/2 check if duplicate entries found</td>
<td></td>
</tr>
<tr>
<td>+1/2 if duplicates, copy only one to result (without use of additional structure)</td>
<td></td>
</tr>
<tr>
<td>+1/2 update indices into both arrays (list1 and list2)</td>
<td></td>
</tr>
<tr>
<td>+1 nonduplicate entries</td>
<td></td>
</tr>
<tr>
<td>+1/2 copy only smallest entry to result (without use of additional structure)</td>
<td></td>
</tr>
<tr>
<td>+1/2 update index into that array only (list1 or list2)</td>
<td></td>
</tr>
</tbody>
</table>

Note: Solution may use constants as returned from part A.

Usage: -1/2 compareTo instead of compareCustomer for Customer objects
Question 4: Drop Game (MBS)

Part A: dropLocationForColumn  3 1/2 points

+1 1/2 loop over Locations in column
  +1/2 correct loop (traverse entire column or until empty location found)
  +1 construct Location object in context of loop
    +1/2 attempt using column
    +1/2 correct

+1 1/2 find drop Location
  +1/2 check if constructed Location is empty
  +1 if exists, return empty Location with largest row # (no loop, no point)

+1/2 return null if column is full

Part B: dropMatchesNeighbors  5 1/2 points

+1 get drop Location
  +1/2 attempt (must call dropLocationForColumn)
  +1/2 correct (must use result)

+1/2 return false if drop location is null

+1 1/2 get neighboring pieces
  +1/2 attempt to access adj. neighbors
    (getNeighbor or neighborsOf or row/column access)
  +1/2 correctly access 3 E/W/S neighbor Location objects
  +1/2 correctly access 3 neighbor Piece objects

+2 1/2 determine matches
  +1/2 correct null neighbor test
  +1 compare colors of pieces
    +1/2 attempt (must reference pieceColor)
    +1/2 correct
  +1 return correct Boolean value

Usage: -1 environment or missing theEnv
Most common usage errors are addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem, even if it occurs on different parts of a problem.

<table>
<thead>
<tr>
<th>Nonpenalized Errors</th>
<th>Minor Errors (1/2 point)</th>
<th>Major Errors (1 point)</th>
</tr>
</thead>
<tbody>
<tr>
<td>spelling/case discrepancies*</td>
<td>confused identifier (e.g., len for length or left() for getLeft())</td>
<td>extraneous code which causes side-effect, for example, information written to output</td>
</tr>
<tr>
<td>local variable not declared when any other variables are declared in some part</td>
<td>no local variables declared</td>
<td>use interface or class name instead of variable identifier, for example Simulation.step() instead of sim.step()</td>
</tr>
<tr>
<td>default constructor called without parens; for example, new Fish;</td>
<td>new never used for constructor calls</td>
<td>use private data or method when not accessible</td>
</tr>
<tr>
<td>use keyword as identifier</td>
<td>void method or constructor returns a value</td>
<td>use of object reference that is incorrect, for example, use of f.move() inside method of Fish class</td>
</tr>
<tr>
<td>[r,c], (r)(c) or (r,c) instead of ([r][c]) = instead of == (and vice versa)</td>
<td>modifying a constant (final)</td>
<td>destruction of data structure (e.g., by using root reference to a TreeNode for traversal of the tree)</td>
</tr>
<tr>
<td>length/size confusion for array, String, and ArrayList, with or without ()</td>
<td>use equals or compareTo method on primitives, for example int x; _x.equals(val)</td>
<td>use class name in place of super either in constructor or in method call</td>
</tr>
<tr>
<td>private qualifier on local variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extraneous code with no side-effect, for example a check for precondition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>common mathematical symbols for operators (x • ‡ ≤ ≥ ‹ › ≠)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing { } where indentation clearly conveys intent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing ( ) on method call or around if/while conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing ;s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing “new” for constructor call once, when others are present in some part</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing downcast from collection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing int cast when needed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing public on class or constructor header</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: Spelling and case discrepancies for identifiers fall under the "nonpenalized" category as long as the correction can be unambiguously inferred from context. For example, "Queu" instead of "Queue". Likewise, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the context allows for the reader to assume the object instead of the class.
Question 1: Daily Schedule

PART A:

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

PART B:

```java
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

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

```java
public double purchasePrice()
{
    return (1 + taxRate) * getListPrice();
}
```

PART B:

```java
public class Vehicle extends TaxableItem
{
    private double dealerCost;
    private double dealerMarkup;

    public Vehicle(double cost, double markup, double rate)
    {
        super(rate);
        dealerCost = cost;
        dealerMarkup = markup;
    }

    public void changeMarkup(double newMarkup)
    {
        dealerMarkup = newMarkup;
    }

    public double getListPrice()
    {
        return dealerCost + dealerMarkup;
    }
}
```
Question 3: Customer List

PART A:

```java
public int compareCustomer(Customer other) {
    int nameCompare = getName().compareTo(other.getName());
    if (nameCompare != 0) {
        return nameCompare;
    } else {
        return getID() - other.getID();
    }
}
```

PART B:

```java
public static void prefixMerge(Customer[] list1, Customer[] list2, Customer[] result) {
    int front1 = 0;
    int front2 = 0;

    for (int i = 0; i < result.length; i++) {
        int comparison = list1[front1].compareCustomer(list2[front2]);
        if (comparison < 0) {
            result[i] = list1[front1];
            front1++;
        } else if (comparison > 0) {
            result[i] = list2[front2];
            front2++;
        } else {
            result[i] = list1[front1];
            front1++;
            front2++;
        }
    }
}
```
PART A:

```java
public Location dropLocationForColumn(int column) {
    for (int r = theEnv.numRows()-1; r >= 0; r--)
    {
        Location nextLoc = new Location(r, column);
        if (theEnv.isEmpty(nextLoc))
        {
            return nextLoc;
        }
    }
    return null;
}
```

ALTERNATE SOLUTION

```java
public Location dropLocationForColumn(int column) {
    int maxRow = -1;
    for (int r = 0; r < theEnv.numRows(); r++)
    {
        if (theEnv.isEmpty(new Location(r, column)))
        {
            maxRow = r;
        }
    }
    if (maxRow < 0)
    {
        return null;
    }
    return new Location(maxRow, column);
}
PART B:

public boolean dropMatchesNeighbors(int column, Color pieceColor) {
    Location loc = dropLocationForColumn(column);
    if (loc == null) {
        return false;
    }
    Piece n1 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.WEST)));
    Piece n2 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.EAST)));
    Piece n3 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.SOUTH)));
    return (n1 != null && n1.color().equals(pieceColor) &&
            n2 != null && n2.color().equals(pieceColor) &&
            n3 != null && n3.color().equals(pieceColor));
}

ALTERNATE SOLUTION

public boolean dropMatchesNeighbors(int column, Color pieceColor) {
    Location loc = dropLocationForColumn(column);
    if (loc == null) {
        return false;
    }
    ArrayList neighbors = theEnv.neighborsOf(loc);
    int colorCount = 0;
    for (int i = 0; i < neighbors.size(); i++) {
        Piece nextNbr = (Piece)(theEnv.objectAt((Location)neighbors.get(i)));
        if (nextNbr != null && nextNbr.color().equals(pieceColor)) {
            colorCount++;
        }
    }
    return (colorCount == 3);