```java
/**
 * Linked List is a collection of data nodes.  All methods here relate to
 * how one can manipulate those nodes.
 *
 * As a student at Union College, I am part of a community that values
intellectual effort, curiosity and discovery. I understand that in order to
truly claim my educational and academic achievements, I am obligated to act
with academic integrity. Therefore, I affirm that I will carry out my
academic endeavors with full academic honesty, and I rely on my fellow
students to do the same.
 *
 * @author Blair Hagen
 * @version 5-5-2016
 */
public class LinkedList
{
    private int length;          // number of nodes
    private ListNode firstNode; // pointer to first node

    public LinkedList()
    {
        length=0;
        firstNode=null;
    }

    /** getter
     *
     * @return number of nodes in the list
     */
    public int getLength() {return length;}


    /** insert new Event at linked list's head
     *
     * @param newData the Event to be inserted
     */
    public void insertAtHead(AgendaItem newData)
    {
      ListNode newnode = new ListNode(newData);
        if (getLength() == 0)
        {
            firstNode=newnode;
        }
        else
        {
            newnode.next=firstNode;
            firstNode=newnode;
        }
        length++;
    }

    /**
     * @return a string representation of the list and its contents.
     */
    public String toString()
    {
      String toReturn="(";
```

```
      ListNode runner;
      runner = firstNode;
      while (runner != null) {
            toReturn = toReturn + runner;
            runner = runner.next;
            if (runner != null) {
                toReturn = toReturn + ",\n";
            }
      }
      toReturn = toReturn + ")";
      return toReturn;
    }

    /**
     * insert new Event into sorted position in LL
     *
     * @param newData the Event to insert
     */
    public void insertSorted(AgendaItem newData)
    {
        ListNode nodeBefore = this.findNodeBefore(newData);
        if (nodeBefore == null) {
            insertAtHead(newData);
        }
        else {
            insertAfter(nodeBefore, newData);
        }
    }

    /**
     * Given a new event to be inserted in the list, finds the correct
position for it.
     *
     * @param newData an event to be inserted in the list
     *
     * @return a pointer to the node in the linked list that will
     * immediately precede newData once newData gets inserted.
     * Returns null if no such node exists (which means newData goes first).
     */
    private ListNode findNodeBefore(AgendaItem newData)
    {
      if (getLength() == 0)
      {
            return(null);
      }
      else
      {
            ListNode runner = firstNode;
            if (runner.data.compareTo(newData) == 1)
            {
                  return(null);
            }
            else
            {
                  while (runner.next != null &&
runner.next.data.compareTo(newData) == -1)
                  {
```

```
                        runner = runner.next;
                    }
                    while (runner.next != null &&
runner.next.data.compareTo(newData) == 0)
                    {
                        runner = runner.next;
                    }
                    return(runner);
                }
            }
        }


        /**
         * Given an event to insert and a pointer to the node
         * that should come before it, insert the new event after nodeBefore.
         * Precondition: length >= 1
         *
         * @param nodeBefore the node (already in the list) that should
         * immediately precede the node with newData in it
         * @param newData the event to be inserted after nodeBefore
         */
        private void insertAfter (ListNode nodeBefore, AgendaItem newData)
        {
          ListNode newNode = new ListNode(newData);

          ListNode runner = firstNode;
          while (!runner.equals(nodeBefore))
          {
                runner = runner.next;
          }
          newNode.next = runner.next;
          runner.next = newNode;

          length++;
        }
    }
```

```
/**
 * An Event is a gathering that happens on a particular date.
 * It has a defined name, starting time, and ending time.
 *
 * @authors Chris Fernandes and Blair Hagen
 * @version 5/5/2016
 */
public class Event implements AgendaItem
{
    private String name;    // name/description of event
    private int year;       // 4-digit year in which event occurs (e.g. 2011)
    private int month;      // number of month 1=Jan, 12=Dec
```

```
    private int day;         // day of the month
    private int startTime;   // time the event starts in military time
                                    // (e.g. 1:15PM is 1315 and 9:00AM is
900)
    private int endTime;     // time the event ends in military time

    /**
     * Non-default constructor.
     *
     * @param eventName name of event
     * @param year  4-digits
     * @param month integer: 1=Jan, 2=Feb, etc.
     * @param day day of the month
     * @param start start time in military time
     * @param end end time in military time
     */
    public Event(String eventName,
                 int year, int month, int day,
                 int start, int end)
    {
        this.name=eventName;
        this.year=year;
        this.month=month;
        this.day=day;
        this.startTime=start;
        this.endTime=end;
    }

    /**
     * Each of the following getters returns the eponymous part of the event
     * @return name, year, month, day, startTime, or endTime
     */
    public String getName() {return name;}
    public int getYear() {return year;}
    public int getMonth() {return month;}
    public int getDay() {return day;}
    public int getStart() {return startTime;}
    public int getEnd() {return endTime;}

    /**
     *  @return returns the event as a printable String
     */
    public String toString()
    {
        return getName() + "  "
                + getMonth() + "/" + getDay() + "/" + getYear() + "  "
                + getStart() + "-" + getEnd();
    }

    /**
     * Compares this event with otherEvent (agendaItem) to see which one is
earlier.
     *
     * @param otherEvent the agendaItem to compare with
     * @return 1 if this event is later than otherEvent,
     * -1 if this event is earlier than otherEvent,
     * or 0 if the two events occur on exactly the same
```

```
     * day with the exact same START time.
     */
    public int compareTo(AgendaItem otherEvent)
    {
      if (getYear() > otherEvent.getYear()) {
            return 1;
      } else if (getYear() < otherEvent.getYear()) {
            return -1;
      } else if (getMonth() > otherEvent.getMonth()) {
            return 1;
      } else if (getMonth() < otherEvent.getMonth()) {
            return -1;
      } else if (getDay() > otherEvent.getDay()) {
            return 1;
      } else if (getDay() < otherEvent.getDay()) {
            return -1;
      } else if (otherEvent instanceof Event && getStart() > ((Event)
otherEvent).getStart()) {
            return 1;
      } else if (otherEvent instanceof Event && getStart() < ((Event)
otherEvent).getStart()) {
            return -1;
      } else {
            if (otherEvent instanceof Reminder)
            {
                  return 1;
            }
            else
            {
                  return 0;
            }

      }
    }
}
```

```
/**
 * A reminder is similar to an Event but it
 * lacks a start and end time but still has a
 * date.
 *
 * @author Chris Fernandes and Blair Hagen
 * @version 5/5/16
 */
public class Reminder implements AgendaItem
{
    private String msg;
    private String date;
```

```java
private static final int NUM_PARTS = 3;
private static final int MONTH_PART = 0;
private static final int DAY_PART = 1;
private static final int YEAR_PART = 2;

/**
 * Creates a reminder
 *
 * @param msg The text of the reminder
 * @param date The date of the reminder (in mm/dd/yyyy format)
 */
public Reminder(String msg, String date)
{
    this.msg = msg;
    this.date = date;
}

/**
 * @return the Reminder as a String
 */
public String toString()
{
    return msg + " "  + date;
}

/**
 * @return the month of the reminder
 */
public int getMonth()
{
    return getDateParts()[MONTH_PART];
}

/**
 * @return the day of the reminder
 */
public int getDay()
{
    return getDateParts()[DAY_PART];
}

/**
 * @return the year of the reminder
 */
public int getYear()
{
    return getDateParts()[YEAR_PART];
}

/**
 * Parses the date string to determine the 3 integer values for
 * month, day, and year.
 *
 * @return an array of 3 ints in the order {month, day, year}
 */
private int[] getDateParts()
```

```
    {
        int[] intParts = new int[NUM_PARTS];
        String[] stringParts = date.split("/");

        for (int i = 0; i < stringParts.length; i++) {
            intParts[i] = Integer.valueOf(stringParts[i]).intValue();
        }

        return intParts;
    }

    /**
     * Compares this reminder with otherEvent (agendaItem) to see which one
is earlier.
     *
     * @param otherEvent the agendaItem to compare with
     * @return 1 if this event is later than otherEvent,
     * -1 if this event is earlier than otherEvent,
     * or 0 if the two events occur on exactly the same
     * day with the exact same START time.
     */
    public int compareTo(AgendaItem otherEvent)
    {
      if (getYear() > otherEvent.getYear()) {
            return 1;
      } else if (getYear() < otherEvent.getYear()) {
            return -1;
      } else if (getMonth() > otherEvent.getMonth()) {
            return 1;
      } else if (getMonth() < otherEvent.getMonth()) {
            return -1;
      } else if (getDay() > otherEvent.getDay()) {
            return 1;
      } else if (getDay() < otherEvent.getDay()) {
            return -1;
      } else {
            if (otherEvent instanceof Event)
            {
                  return -1;
            }
            else
            {
                  return 0;
            }
      }
    }
}
```

```java
/**
 * Tests new (and old?) LinkedList methods.
 *
 * @author Blair Hagen, Aaron Cass, and Chris Fernandes
 * @version 5/5/16
 */
public class LinkedListTester
{
    public static void main(String[] args)
    {
        Testing.setVerbose(true);

      Testing.startTests();

      test21();

      test221();

      test2321();

      testSameTime();

      testAnotherSameTime();

      Testing.finishTests();
    }

    private static void test21()
    {
      Testing.testSection("1 -> 21\n"
                  + "Tests inserting at head with one node");

        LinkedList list = new LinkedList();

        Event E1 = new Event("chess club", 2016, 5, 25, 1900, 1930);
        Event E2 = new Event("boy scouts", 2016, 5, 23, 900, 1000);

        list.insertAtHead(E1);
        list.insertSorted(E2);

      Testing.assertEquals("2-node list contents",
                    "(" + E2 + ",\n" + E1 + ")",
                    list.toString());
      Testing.assertEquals("2-node list length", 2, list.getLength());
    }

    private static void test221()
    {
      Testing.testSection("2 -> 21\n"
                  + "Tests inserting at tail with one node");

      LinkedList list = new LinkedList();

        Event E1 = new Event("chess club", 2016, 5, 25, 1900, 1930);
        Event E2 = new Event("boy scouts", 2016, 5, 23, 900, 1000);

        list.insertAtHead(E2);
```

```
        list.insertSorted(E1);

        Testing.assertEquals("2-node list contents", "(" + E2 + ",\n" + E1 +
")", list.toString());
        Testing.assertEquals("2-node list length", 2, list.getLength());
    }

    private static void test2321()
    {
      Testing.testSection("2 -> 321\n"
                  + "Tests inserting at head and tail with two nodes");

        LinkedList list = new LinkedList();

        Event E1 = new Event("chess club", 2016, 5, 25, 1900, 1930);
        Event E2 = new Event("boy scouts", 2016, 5, 23, 900, 1000);
        Event E3 = new Event("janitor", 2016, 5, 22, 0, 1000);

        list.insertAtHead(E1);
        list.insertSorted(E2);
        list.insertSorted(E3);

        Testing.assertEquals("3-node list contents", "(" + E3 + ",\n" + E2 +
",\n" + E1 + ")", list.toString());
        Testing.assertEquals("3-node list length", 3, list.getLength());
    }

    private static void testSameTime()
    {
      Testing.testSection("Tests inserting an event with the same start time
as another");

      LinkedList list  = new LinkedList();

      Event E1 = new Event("chess club", 2016, 5, 25, 1900, 1930);
      Event E2 = new Event("checkers club", 2016, 5, 25, 1900, 1935);

      list.insertAtHead(E1);
      list.insertSorted(E2);

      Testing.assertEquals("2-node list contents", "(" + E1 + ",\n" + E2 +
")", list.toString());
      Testing.assertEquals("2-node list length", 2, list.getLength());
    }

    private static void testAnotherSameTime()
    {
      Testing.testSection("Tests inserting another event at the same time as
the other two events");

      LinkedList list = new LinkedList();

      Event E1 = new Event("chess club", 2016, 5, 25, 1900, 1930);
      Event E2 = new Event("checkers club", 2016, 5, 25, 1900, 1935);
      Event E3 = new Event("magic club", 2016, 5, 25, 1900, 1920);

      list.insertAtHead(E1);
```

```
      list.insertSorted(E2);
      list.insertSorted(E3);

      Testing.assertEquals("3-node list contents", "(" + E1 + ",\n" + E2 +
",\n" + E3 + ")", list.toString());
      Testing.assertEquals("3-node list length", 3, list.getLength());
    }
}
```

```
/**
 * Tests the Interface and the LL which uses it
 *
 * @author Blair Hagen and Chris Fernandes
 * @version 5/5/16
 */
public class InterfaceTester {

      public static void main(String[] args) {
            Testing.setVerbose(true);

      Testing.startTests();

      compareToTests();

      Testing.finishTests();

      insertTests();
      }

      private static void compareToTests() {

        Event E1 = new Event("boy scouts", 2016, 6, 5, 900, 1000);
        Event E2 = new Event("book club", 2016, 5, 10, 1500, 1630);

        Testing.assertEquals("Event-Event different
dates",1,E1.compareTo(E2));

        Reminder R1 = new Reminder("take out the trash", "6/5/2016");
        Reminder R2 = new Reminder("take out the recycling", "6/4/2016");
        Reminder R3 = new Reminder("take out the compost", "6/8/2016");

        Testing.assertEquals("Compare Reminder to Event: Same Day", 1,
E1.compareTo(R1));
        Testing.assertEquals("Compare Event to Reminder: Same Day", -1,
R1.compareTo(E1));

        Testing.assertEquals("Compare Reminder to Event: Event later", 1,
E1.compareTo(R2));
        Testing.assertEquals("Compare Event to Reminder: Event later", -1,
R2.compareTo(E1));
```

```
        Testing.assertEquals("Compare Reminder to Event: Event Earlier", -1,
E1.compareTo(R3));
        Testing.assertEquals("Compare Event to Reminder: Event Earlier", 1,
R3.compareTo(E1));


      }

      private static void insertTests() {

        LinkedList newList = new LinkedList();

            Event E1 = new Event("boy scouts", 2016, 6, 5, 900, 1000);
        Event E2 = new Event("book club", 2016, 7, 10, 1500, 1630);

        Reminder R1 = new Reminder("take out the trash", "6/5/2016");
        Reminder R2 = new Reminder("take out the recycling", "6/4/2016");
        Reminder R3 = new Reminder("take out the compost", "6/8/2016");

        newList.insertAtHead(E1);
        newList.insertSorted(R2);
        Testing.assertEquals("Inserting Reminder earlier than Event", "(" +
R2 + ",\n" + E1 + ")", newList.toString());
        newList.insertSorted(R1);
        Testing.assertEquals("Inserting Reminder on same day as Event", "(" +
R2 + ",\n" + R1 + ",\n" + E1 + ")", newList.toString());
        newList.insertSorted(R3);
        Testing.assertEquals("Inserting Reminder later than Event",  "(" + R2
+ ",\n" + R1 + ",\n" + E1 + ",\n" + R3 + ")", newList.toString());
        newList.insertSorted(E2);
        Testing.assertEquals("Inserting Event later than all", "(" + R2 +
",\n" + R1 + ",\n" + E1 + ",\n" + R3 + ",\n" + E2 + ")", newList.toString());
      }


}
```

```
/**
 * Interface between Event and Reminder classes
 *
 * @author Blair Hagen
 * @version 5-5-2016
 *
 */


public interface AgendaItem {

/**
```

```
 * Returns string representation of agendaItem
 * The output of an Event will have a start/end time,
 * the output of a reminder will not.
 * @return
 *          "name  month/day/year  start-end" If the object is an Event
 *          "name  month/day/year" If the object is a Reminder
 */
public String toString();

/**
 * Returns the month of the agendaItem.
 * Makes no difference if the agendaItem is an event or a reminder
 * @return month
 *          month of agendaItem as an integer
 */
public int getMonth();

/**
 * Returns the day of the agendaItem.
 * Makes no difference if the agendaItem is an event or a reminder
 * @return day
 *          day of agendaItem as an integer
 */
public int getDay();

/**
 * Returns the year of the agendaItem.
 * Makes no difference if the agendaItem is an event or a reminder
 * @return year
 *          year of agendaItem as an integer
 */

public int getYear();
/**
 * Returns -1, 0, or 1 depending on if the agenda item is earlier, later, or
 * at the same time as otherItem. If the two objects are events, dates and
 * start times are used to determine their relationship. If two objects are
 * reminders, just the dates of the two are used. If one object is an event
 * and the other a reminder and they have the same date, the reminder will
 * always come before the event and 0 will never be returned.
 *
 * @param otherItem
 *          the reminder/event to compare this object to
 * @return
 *          -1 if this AgendaItem is earlier than otherItem
 *          0 if the two are concurrent
 *          1 if this AgendaItem is later than otherItem
 */
public int compareTo(AgendaItem otherItem);
}
```

```java
/**
 * ListNode is a building block for a linked list of data items
 *
 * This is the only class where I'll let you use public instance variables.
 * It's so we can reference information in the nodes using cascading dot
 * notation, like
 *          N.next.data instead of
 *          N.getNext().getData()
 *
 * @author C. Fernandes and G. Marten
 * @version 5/4/2016
 */
public class ListNode
{
    public AgendaItem data;       // something to put on your calendar
    public ListNode next;   // pointer to next node

    /** Non-default constructor
     *
     * @param newData a reservation you want stored in this node
     */
    public ListNode(AgendaItem newData)
    {
        this.data = newData;
        this.next = null;
    }

    // if you say "System.out.println(N)" where N is a ListNode, the
    // compiler will call this method automatically to print the contents
    // of the node.  It's the same as saying
"System.out.println(N.toString())"
    public String toString()
    {
        return data.toString();  // call the toString() method in Event class
    }
}
```

searches for an event with a later start time then inserts the event before that one.

```
if calendar is empty
      return null
else
      runner equals the firstNode
      if the newData compared to firstNode equals -1
            runners next equals firstNode
            firstNode equals runner
      while runner doesnt equal null and runner compared to newData equals -1
or 0
            runner equals runners next

return the runner
```

step through the linkedlist until the nodeBefore is found, then make a new node with given
event and insert that node after nodeBefore and connect the new node to the later nodes.

(precondition: the linkedList has at least one node)

runner equals the first node
while the runner doesn't equal the nodeBefore
	set runner equal to the next node

make a new node using the event data
set the runners next node to the new nodes next node
set the runners next node to the new node

1:
	findBeforeNode: 0(n)
	insertAfter: 0(n)

2:
	The insertAfter method deals directly with listNodes, so it should contain those nodes to the LinkedList
	class as that is the only reason why ListNode can have public instance variables. This linkedList class
	is only supposed to deal with sorted events, so the only method the user should need to use is insertSorted
	and insertAtHead to manage the capacity of the list. insertAfter would not conform to that, so it must be hidden.

	findNodeBefore is used to find a specific event and returns a node, which should never exit the LinkedList
	class as stated above because that is what allows LinkNode to have public instance variables.

3:
	A LinkedList implementation would be best for this situation as it is more efficient to add events to a linkedList
	versus adding events to an array. An array requires large amounts of information to be copied and moved around
	while a linkedlist only requires a few pointers to be changed.

4:
	An array implementation would be best for this situation because there are not many items being added/removed,
	and it is easier to access an array using indexes than a linkedlist. To access an index in a linkedlist, you
	must step through the list till you find the index, but with an array the index value can be accessed directly.