Programming Project 4

Due Friday, November 13 at 11:59pm

IMPORTANT: *Read the* ***Do's and Dont's*** *in the* ***Course Honor Policy*** *found on the Canvas class site.*

I. Overview

The purpose of this project is to practice using linked lists, generic types, iterators, and comparable.

The theme of this project is scheduling jobs to maximize profit.

II. Code Readability (20% of your project grade)

New This Project The comments at the top of your class and methods need to be in JavaDoc format. JavaDoc comments are introduced in Lab 11, and there are a number of examples in the class code. You can find lots of info about JavaDoc in Chapter 7 of the "Jave in a Nutshell" book.

To receive the full readability marks, your code must follow the following guideline:

- All variables (fields, parameters, local variables) must be given appropriate and descriptive names.
- All variable and method names must start with a lowercase letter. All class names must start with an uppercase letter.
- The class body should be organized so that all the fields are at the top of the file, the constructors are next, the non-static methods next, and the static methods at the bottom with the main method last.
- There should not be two statements on the same line.
- All code must be properly indented (see Appendix F of the Lewis book for an example of good style). The amount of indentation is up to you, but it should be at least 2 spaces, and it must be used consistently throughout the code.
- You must be consistent in your use of {, }. The closing } must be on its own line and indented the same amount as the line containing the opening {.
- There must be an empty line between each method.
- There must be a space separating each operator from its operands as well as a space after each comma.
- There must be a comment at the top of the file that **is in proper JavaDoc format** and includes both your name and a description of what the class represents. The comment should include tags for the author. (See Appendix J of the Lewis book of pages 226-234 if the Evans and Flanagan book.)
- There must be a comment directly above each method (including constructors) that **is in proper JavaDoc format** and states *what* task the method is doing, not how it is doing it. The comment should include tags for any parameters, return values and exceptions, and the tags should include appropriate comments that indicate the purpose of the inputs, the value returned, and the meaning of the exceptions.
- There must be a comment directly above each field that, in one line, states what the field is storing.
- There must be a comment either above or to the right of each non-field variable indicating what the variable is storing. Any comments placed to the right should be aligned so they start on the same column. Variables that are only used as an index in a loop do not need a comment.
- There must be a comment above each loop that indicates the purpose of the loop. Ideally, the comment would consist of any preconditions (if they exist) and the subgoal for the loop iteration.
- Any code that is complicated should have a short comment either above it or aligned to the right that explains the logic of the code.

III. Program Testing (20% of your project grade)

You are to create JUnit tests. If you added additional methods to the linked list class (including methods you added in a lab), you should submit a *separate* JUnit test class for the linked list that tests those methods. You do not have to write a JUnit tests for the methods you did not write. For the other class or classes of this project, you may submit one or multiple JUnit test classes. It is your choice.

Your report should be a short document that explains why the JUnit tests you provide thoroughly test your program. **However**, you should not JUnit test the `main` method or methods that specifically read from a file or methods that write to the screen. For these methods, your testing report should describe the tests you did to judge the correctness of your program.

IV. Java Programming (60% of your grade)

Very Important Programming Rule

- The main purpose of this project is to gain practice using linked lists, arrays, and iterators. In several places you have your choice of whether you want to use an array. array list, or linked list, and whether to use the linked list we developed in class or the one from the Java API.

  You must use arrays and `ArrayList` explicity as arrays. Therefore you should not use the automatic resizing feature of `ArrayList` and should instead set its initial capacity appropriately.

  You should use the linked list explicity as a linked list. You may not use code that is inefficient in a linked list *even if the Java API may optimize that code*. Therefore, you should not access a linked list element by its index. Instead, you should use methods that add or remove elements from the ends (or just one end in a single linked list), test if the list is empty, or use an iterator. Similarly, you should not add or use inefficient helper methods in our linked list class.

- The second purpose of the homework is to use generic types appropriately. Your code should not generate warning messages for missing geeneric types. (You may have to turn on the compiler option that shows these warnings.) You should defnintely not add the @SuppressWarnings annotation to your program. There is one exception in that if you are using arrays, it is impossible to avoid a warning message if you need to typecast an array to a generic type.

Part I: Doubly Linked Lists

You are to expand the `DoubleLinkedList` class you worked on in lab. You are to make the `DoubleLinkedList` implement `Iterable` and have the `iterator` method return a `ListIterator` from the API. `ListIterator` is an extension of `Iterator` and has additional method stubs that must be overridden. You are required to implement the methods

1. `add`
2. `hasNext`
3. `hasPrevious`
4. `next`
5. `previous`
6. `remove`

Your implementations *must* match the descriptions given in the Java API. You *may* implement the other methods of the interface, but that is not required. For any method you choose to not implement, have the method throw a `UnsupportedOperationException`.

Part II: Job Scheduling

For the rest of this project, you will write a program that schedules jobs. Job scheduling is an extremely important problem in computer science, and depending on the specifications of the problem, there are very different optimal algorithms. For this homework, each job has the following properties and the goal is to schedule the jobs in order to maximize profit.

- *earliest start time* the earliest time at which the job may be started
- *deadline* the time by which the job must be completed
- *duration* the length of time it will take to do the job
- *profit* the amount of money you will make if you complete the job

We will assume that you can only work on one job at a time, and you must complete a job that you start before you can begin the next one.

For example, consider the following three jobs:

Job #	Earliest Start	Deadline	Duration	Profit
1	2	6	2	10
2	3	6	3	20
3	3	9	5	15

Job 1 must be completed by time 6 and has duration 2. So, Job 1 can be started at time 2, 3, or 4. Job 2 must be completed by time 6 and has duration 3. So, Job 2 must be started at time 3. Note that it is impossible to schedule both Job 1 and Job 2. Job 2 must use times 3-5, but Job 1 has the option of using times 2-3, 3-4, or 4-5. Job 3 must be completed by time 8 and has duration 5, so Job 3 may either start at time 3 and use times 3-7, or it may start at time 4 and use times 4-8. In this case, the most profit can be achieved from scheduling Job 1 to start at time 2. Job 1 will complete by time 4 so we can schedule Job 3 to start at time 4, and we have a total profit of 25. The second best schedule is to schedule only Job 2 at time 3 and get a profit of 20.

The Job Classes

- The `Job` class
  A `Job` consists of five values:

  1. `id` a unique value that indentifies the job
  2. `earliestStart` the earliest time at which the job may be started
  3. `deadline` the latest time at which the job must complete
  4. `duration` the length of time it takes to complete the job
  5. `profit` the amount of profit to be earned if the job is completed

  All the values are type `int`. Create fields, getter methods for the fields, a constructor that initializes all the fields, and an appropriate `equals` method.

  The `Job` should be `Comparable` with the default ordering by `id` from smallest to largest.

  You should create at least two `Comparators` for `Job`. One should sort jobs by `earliestStart`, from smallest to largest, and one should sort jobs by `profit` from largest to smallest. Have the methods `getStartComparator` and `getProfitComparator` return these `Comparators`.

- The `CompoundJob` class
  A `CompoundJob` *is a* job that consists of "sub"-jobs. Each subjob is type `Job`. (You should decide if the subjobs should be stored in an array, ArrayList, or linked list.) In a `CompoundJob` all the subjobs should be completed in order to receive the profit from the job. The subjobs must be completed in order, but do not have to be scheduled consecutively. The `earliestStart` of the `CompoundJob` is the `earliestStart` of the first subjob. The `duration` of the `CompoundJob` is the sum of the durations of each subjob. The `deadline` of the `CompoundJob` is the deadline of the last subjob. The `earliestStart` and `deadline` for each subjob (other than the earliest start of the first subjob and deadline of the last subjob) can be calculated appropriately. The profit of the `CompoundJob` is only earned if all subjobs are successfully completed. We will assume that the subjobs do not have any profit (though you may change this for the extra credit).

  Besides the methods it inherits, the `CompoundJob` class should have the method

  1. `getSubJobs()` retrieves the a list (the type is up to you) of the subjobs of the `CompoundJob`

  and the constructor

  1. `CompoundJob(int profit, Job... subJobs)`
     `profit` is the profit earned if all the subjobs of the compound job can be scheduled and completed. `Job...` is a Java shortcut called a *variable length parameter*, and it is a shortcut for an array. The type of `subJobs` is `Job[]`, but you have two ways to pass the array into the constructor, either as an array or as the individual elements separated by commas. `new CompoundJob(jobsArray)` or `new CompoundJob(job1, job2, job3, job4)`.

  For example, supposed the compound job consists of three subjobs, the first subjob has *earliest start* 1 and *duration* 6, the second subjob has *duration* 3, and the third subjob has *duration* 8 and *dealine* 25. The compound job should be set to have *earliest start* 1, *duration* 17, and *deadline* 25. In addition, to help you schedule the subjobs along with all the other jobs, I recommend that you set the *deadline* for the first subjob to be 14 (that is the latest it can finish such that the other two subjobs can be scheduled), the second subjob should have its *deadline* set to 17 and its *earliest start* set to 7, and the third subjob should have its *earliest start* set to 10.

The Scheduling Classes

- The `ScheduleSlot` class
  A `ScheduleSlot` consists of two values:

  1. `job` the job that is scheduled in this slot

2. `startTime` the start time of this slot

`startTime` should type `int` and `job` should be type `Job`. Create fields, getter methods for the fields, a setter method for `startTime` and a constructor that initializes both fields. The `ScheduleSlot` will represent the scheduled time for the `job`. We will start the job at time `startTime` and work on the job exclusively for the length of the job's `duration`.

- The `Schedule` type
  A Schedule is an (ordered) doubly linked list of `ScheduleSlots`.
  There are multiple ways you can define the `Schedule`, and you are free to choose an appropriate one. While you are encouranged to use the `DoubleLinkedList` class you developed in Part I, you may use the `LinkedList` class from the API. (Note that the Schedule cannot be an array or an ArrayList because we will be doing lots of insertions into the middle of the list.) The slots will be ordered by `startTime`.

- The `ScheduleMetric` interface
  The interface will have one method stub `scheduleJob` that takes a `Schedule` and a `Job` as input and returns a `boolean`.

- The `ScheduleAsLateAsPossible` class
  `ScheduleAsLateAsPossible` is a `ScheduleMetric`. The `scheduleJob` method attempts to place the job into the schedule such that the job will be completed at as late a time as possible given the job's `deadline` and the other jobs in the schedule. A `ScheduledSlot` containing the job should only be added to the schedule if the job's scheduled time (the `ScheduledSlot`'s `startTime` to the `startTime` plus the job's `duration`) starts no earlier than the job's `earliestStart`, completes by the job's `deadline`, and the scheduled time does not overlap the scheduled times of any other jobs in the schedule. Scheduling as late as possible means that it is impossible to have a `ScheduleSlot` with a later `startTime` without having the job's scheduled time exceeding the job's `deadline` or overlapping with some other job on the schedule. If it is possible to schedule the job, a new `ScheduleSlot` containing the job and the necessary start time is added to the schedule and the method returns `true`. Otherwise, the method returns `false`. As a hint, try iterating through the `ScheduleSlots` starting at the last scheduled job.

  If the job is a `CompoundJob`, you should schedule each of the subjobs individually, but if it is impossible to schedule all the subjobs, you must remove the `ScheduleSlotss` of any subjobs that were placed in the schedule. As a hint, traverse the subjobs in the same order that you are traversing through the schedule so that you do not have to restart the iterations with each subjob.

- The `ScheduleAsEarlyAsPossible` class
  `ScheduleAsEarlyAsPossible` is a `ScheduleMetric`. The `scheduleJob` method attempts to place the job into the schedule such that the job will be completed at as early a time as possible given the job's `startTime` and the other jobs in the schedule. A `ScheduledSlot` containing the job should only be added to the schedule if the job's scheduled time (the `ScheduledSlot`'s `startTime` to the `startTime` plus the job's `duration`) starts no earlier than the job's `earliestStart`, completes by the job's `deadline`, and the scheduled time does not overlap the scheduled times of any other jobs in the schedule. Scheduling as early as possible means that it is impossible to have a `ScheduleSlot` with an earlier `startTime` without having the `startTime` earlier than the job's `eariestStart` or overlapping with some other job on the schedule. If it is possible to schedule the job, a new `ScheduleSlot` containing the job and the necessary start time is added to the schedule and the method returns `true`. Otherwise, the method returns `false`. As a hint, try iterating through the `ScheduleSlots` starting at the first scheduled job.

  If the job is a `CompoundJob`, see the note for how to handle compound jobs in `ScheduleAsLateAsPossible`.

The Job Scheduler

- The `JobScheduler` class
  A class that does the job scheduling. It should have the following methods:

  1. `scheduleJobs` takes a list of `Jobs` (you should decide if this should be an array, ArrayList, or linked list), a `Comparator` for the jobs, and a `ScheduleMetric`. The method should sort the list of `Jobs` using the `Comparator`, create an initially empty `Schedule`, and then the iterate through the list (try using a *foreach* loop) and attempt to schedule each job using the `ScheduleMetric`. The `Schedule` is returned.

     If you are using the a linked list implementation from class, use the fast *merge sort* algorithm we will develop in lecture. Do *not* use `insertInOrder` to sort the list because it is much too slow.

  2. `main` the main method will accept a command line argument that is the filename for a file of job data. The method should:

     1. open the file
     2. create a list of jobs from the data in the file (you decide whether this should be an array, ArrayList or linked list and see below for the format of the job file). I recommend using the `Scanner` class from the API and its `nextInt` method.
     3. create a schedule that considers the jobs in order by profit, starting with the largest profit, and schedules each job as late as possible
     4. create a schedule that considers the jobs in order by profit, starting with the largest profit, and schedules each job as early as possible
     5. prints each schedule in an appropriate format including the total profit of each schedule and a message indicating which schedule has more profit

     The main method should catch errors and print appropriate messages instead of crashing.

     Here are two other methods you can add you can add to your `JobScheduler` class if you want a convenient way to create lists and files of random jobs to help you with your testing

  3. `createRandomJobs` creates an array of random jobs. Note that the method does not create `CompoundJobs`.

```
/**
 * Creates a file with random job data, based on the parameters given.  The job data will be
 * sorted by earliest start time.
 * @param numJobs   the number of jobs to create
 * @param maxStart  the latest time at which a job may set as its earliest start time
 * @param maxDuration  the maximum time that a job can take to complete
 * @param maxLag   the greatest time between the earliest start time for a job and the latest time that a job must start to meet :
 * @param maxProfic  the maximum amount of profit from a job
 * @return an array containing the random jobs
 */
public Job[] createRandomJobs(int numJobs, int maxStart, int maxDuration, int maxLag, int maxProfit) {
  Job[] jobs = new Job[numJobs];

  // For each desired job, create 4 random numbers based on the parameters, use the numbers to create a job,
  // and store the job in an array.
  for (int i = 0; i < numJobs; i++) {
    int start = (int)(Math.random() * maxStart) + 1;
    int duration = (int)(Math.random() * maxDuration) + 1;
    int deadline = start + duration + (int)(Math.random() * (maxLag + 1));
    int profit = (int)(Math.random() * maxProfit) + 1;
    jobs[i] = new Job(i, start, deadline, duration, profit);
```

```
    }

    // Sort the jobs by earliest start time
    Arrays.sort(jobs, Job.getStartComparator());

    return jobs;
  }
```

4. `createJobFile`: creates a file with job data.

```
  /**
   * Creates a file with job data.
   * @param fileName  the name of the file to store the job data.  The file must not exist.
   * @param jobs an array containing the jobs
   */
  public void createJobFile(String fileName, Job[] jobs) {
    // Check that the output file does not already exist
    File file = new File(fileName);
    if (file.exists()) {
      System.out.println("Error: file " + fileName + " already exists.");
      return;
    }

    try {
      BufferedWriter writer = new BufferedWriter(new FileWriter(file));

      // For each job in the array, write the job to the file as 5 numbers separated by spaces.
      // Place an each job on a new line.
      for (int i = 0; i < jobs.length; i++) {
        String s = i + " " + jobs[i].getEarliestStart() + " " + jobs[i].getDeadline() + " " + jobs[i].getDuration() + " " + jobs[i].
        writer.write(s, 0, s.length());
        writer.newLine();
      }
      writer.flush();
      writer.close();
    }
    catch (IOException e) {
      System.out.println("Error writing to file " + fileName);
    }
  }
```

The Job file

A job data file is a list of numbers, five integers on each line, that match the data given in the above table. So, a job file that contains those three jobs will have the data:

```
1 2 6 2 10
2 3 6 3 20
3 3 9 5 15
```

For example, here is a job file joblist.txt that contains 100 jobs and no compound jobs. The files produced by `createJobFile` have the same format. A compound job is identified by multiple jobs with the same job id. (Please note: if you preview the file in Canvas, Canvas adds periods at the start of the lines. Those periods are not actually in the file.)

Extra Credit: (up to a 10% bonus)

The extra credit for this homework is to try to come up with improved algorithms for scheduling the jobs and/or variations that handle compound jobs. You are to code these algorithms and run experiments to determine the performance of these algorithms. If you do the extra credit, you should submit a written report that describes your new algorithms, the results of your experiments, and your analysis. The receive extra credit, you need to create non-trivial algorithms that have a chance of performing better than the above algorithms, and you should provide non-trivial analysis of your results.