## Overview

### Goals

The goals of this assignment are to:

* Understand, implement, and use linked chains of nodes.
* Write classes that implement Java interfaces.
* Gain experience working with Java references.
* Implement exceptions and understand the difference between checked and unchecked exceptions.

### Description

In this assignment, the objective is to create a game where the player will attempt to maximize their score, within a time limit. In the game, there is a never-ending stream of tasks that need to be completed. Each task requires a certain amount of time, each worth a varying number of points. The goal of the game is to try to maximize your score in the allotted time.

## Specifications

Job Class

This class is provided for you and is used to represent a Job that will be used with the list. Each job has a total duration, current number of steps it's been worked on, a name, and a score that it's worth. When the number of steps it's been worked on is equivalent to the total duration, the job should be considered completed.

JobList Class +

This class stores all currently active jobs. It implements ListADT<Job> interface as presenting in the readings and lecture notes.  Note: The get and remove methods throw the IndexOutOfBoundsException if the specified pos is invalid.

The JobList class must be implemented as a **singly-linked chain of Listnode<Job> nodes with a header node and without a tail reference**.  The Listnode<E> class is provided for you in readings and lecture notes.

Note: The ListADT interface in this program **extends Iterable**, so your JobList class must also implement the methods required by Java's java.lang.Iterable interface.

Iterating over job lists is done with a **JobListIterator** object (See next class description).

JobListIterator Class +

Instances of this class are used to iterate through a chain of nodes.  The iterator has direct access to the JobList’s chain of nodes. It's constructor accepts (requires) a Listnode<Job> that is a header node.  This type implements the methods defined in Java's java.lang.Iterator interface, but it does not support the remove() operation.

Note: The very first time the next() method of the resulting iterator is called, it returns the first available data item of the JobList.  If next() is called when the JobListIterator has no job items remaining, the NoSuchElementException should be thrown.

Scoreboard Class

This class implements the ScoreboardADT and contains a single ListADT<Job> that contains only jobs that have been fully completed. Jobs should only be added in at the end of this list, there also exists a method for printing out the current status of the Scoreboard object. Use this class to display the total score accumulated at the end of the game.

JobSimulator Class

This class has been created and provided for you. It contains one important public method, simulateJobs, which takes a ListADT<Job> and the amount of time remaining in the game, and appropriately adds new jobs randomly to the end of the list. The class that implements ListADT<Job> must have the add method fully implemented in order for this function to behave as intended.

Game Class

The Game class is responsible for maintaining the active list of jobs and utilizes the JobSimulator class to create new jobs to be added to the end of the job listing. Functionality must also be present allowing for addition of jobs to the list, and updating individual jobs, in addition to the standard setters and getters.

GameApp Class

This is the main class that starts the program execution and starts the main program loop. It takes two command line arguments, a seed for the random number generator and a positive integer indicating the duration. If either entry is not a positive integer the program should simply terminate.

**The following steps describe the basic program loop.  This loop is repeated while the game is not over.  The game is over when the player runs out of time.**

1. Display how much time remains in the game
2. Use the Game object to create new jobs **(This should only be done on the first iteration, or if a job is successfully executed either to completion or reinsertion into the list, otherwise skip this step)**
3. Use the Game object to display all jobs
4. Prompt the user for an index of a job to work on. There is a time penalty for picking a job that is not at index 0.
   1. The time penalty is equal to the job index selected.
   2. Deduct this time penalty from the total game time remaining.
5. Prompt the user for an amount of time to work on this job
6. Get the job stored at the given index and attempt to work on it for the given amount of time (error checking)
7. Update the job for the specified amount of time
8. If the job has not been completed yet
   1. Prompt the user for an index to insert the element back into the list.
   2. Calculate the time penalty. There is a time penalty for not inserting at position 0.   The time penalty is equal to the index chosen or the total number of items on the list if the index is invalid.
   3. Deduct the time penalty from the total game time remaining.  
      **For example**: If you choose to place the job at position 0, there is no time penalty, but if you choose to place job at index 5 there is a time penalty of 5 that must be subtracted from the total game time remaining.  If you choose index -1, the time penalty is equal to the total number of jobs in the list.
   4. Insert the job back in the list at the specified location if it’s a valid index, else at the end of the list in case of invalid index.
9. Else
   1. Display a success message to the user and update them on their current score.

All places where user input is required must have error checking. Use the getIntegerInput() method as provided to help with this.  After the time left in the game runs out the program should print one final message displaying their final score.

### Input/Output

See example runs for the required prompts and expected input from the user.

* The GameApp class has a class (static) constant Scanner connected to the keyboard input.
* The GameApp class also has a getIntegerInput(prompt) method.  Use this method to get integer input from the user.
* The Job class has a toString() method which returns a string of text as needed for the job to be displayed in the Job listing.  Use this method when outputting jobs as part of job lists.
* The sample runs were run with command line arguments [20, 50](https://canvas.wisc.edu/courses/23073/files/807316/download), and [50 50](https://canvas.wisc.edu/courses/23073/files/807317/download) respectively.
* Read the Javadocs carefully to ensure that the output and input prompts are coming from the correct methods.

Corner Cases

In the event that the amount of time the job is intended to be worked on exceeds the remaining time of the game, the job must be progressed for at most the amount of time remaining in the game. Simply reducing the parameter to that value will be sufficient.

In the event of a job not being completed, but enough time was used where the game would be over, the user is still prompted to reinsert back into the list before the program terminates.

In the event that the amount of time a job is to be worked on exceeds the amount of time that it still requires, it is then fully completed and any remaining time is added back to the game time as it was not used.

### All Classes

The javadocs for files is under construction.  Check back tomorrow for update.   
You are expected to complete those sections of it marked as "//TODO".

You may **not** add any other public methods than those listed in the provided files.

You may **not** modify any class in any way except where noted as "//TODO".

## Steps

After you have read this program page and given thought to the problem we suggest the following steps:

1. **Download this**[**p2.zip**](https://canvas.wisc.edu/courses/23073/files/807245/download)**file to a programming assignment p2 folder that you make.** Unzip and place the files in your p2 program folder in the default unnamed package.  Move the files if necessary to get them in the default package.
2. **Incrementally**implement and thoroughly test your **JobList** class, **JobListIterator** and any additional supporting classes.
3. **Incrementally**implement the **Game** and **GameApp** Test each step to ensure your program is working correctly before implementing the next step.
4. If you are not using the lab computers to develop your program, make sure you compile and run your program to ensure that it works on the Linux lab computers. You can compile your Java source using javac in a terminal window as in this example:
   * javac \*.java

and then run your program using java as in:

* java GameApp <seed> <timeToPlay>

1. Submit your work for grading. We will test each of these classes independently of the other classes to the extent we are able to, as well as test your overall application:
   * JobList class
   * JobListIterator class
   * Scoreboard class
   * Game class
   * GameApp class
   * p2\_task\_list.txt (Submit this file to [Team Building 102](https://canvas.wisc.edu/courses/23073/assignments/63226) assignment)