JAVA PROGRAMMING LANGUAGE - 36B
BERKELEY CITY COLLEGE

LAB A

All classes must follow all good design and coding practices, including good design of classes, code indenting, appropriate comments, and readable and meaningful output.

For each of the programming assignment please make sure to create a Tester class that tests and shows that your program code has been properly tested and verified. Make sure to include the test output below the Tester class code in a commented section.

A. Write a class `CalendarDay` that creates a valid date and stores the month, day, and year in three separate instance variables inside the object.

   The class should contain the following methods:

   - A constructor that takes String data in the form "month/day/year" and initializes the instance variables. For example, new `CalendarDay("12/25/2000")` should create a valid instance of the `CalendarDay` with month = 12, day = 25, and year = 2000.
   - A constructor that takes three separate integer parameters month, day, and year and initializes the instance variables. For example, `new CalendarDay(11,20,2016)` should create an valid instance of the `CalendarDay` with month = 11, day = 20, and year = 2016.
   - A copy constructor
   - Accessor and mutator methods for month, day, and year.
   - Override the `equals` method that compares the current object with another object and return the correct results.
   - A `toString` method that returns a `String` description of the object.

   Additional requirements for the class:

   - The months should be implemented as a group of enumerable objects, with each month verifying the validity of the day based on year and month,
   - The `CalendarDay` should implement the `Comparable` interface and return the correct results when compared to another `CalendarDay` object.
   - You should define three exception classes, one called `MonthException`, another called `DayException`, and a third called `YearException`. The class constructors should check the validity of the input and throw the appropriate exceptions. If the month supplied is anything other than a legal month number (integers from 1 to 12), your class should throw a `MonthException`. Similarly, if the day is not a valid day number (integers from 1 to either 28, 29, 30, or 31, depending on the month and year), then your class should throw `DayException`. If the year is not in the range 1000 to 3000 (inclusive), then the class should throw a `YearException`. (There is nothing very special about the numbers 1000 and 3000 other than giving a good range of likely dates.)
   - The mutator methods should throw an exception if the parameters are invalid or changes the object to an inconsistent state. When changes are made to the month or day or year, it

should verify to make sure that it is a valid day. If not, these methods should throw one of the above exception with a meaning message.

Create a Tester class and test all the methods, including throwing and catching exceptions. Use a loop and the try-catch statements in the main method that gives the user an opportunity to re-enter the month, day, and date when an exception is thrown by the `CalendarDay` class constructors.

Also, create an array of random `CalendarDay` objects and use the `Arrays.sort` method to sort these objects.

B. Write a `StockExchange` class that manages only one stock and communicates the changes in the stock price to a list of stock exchange customers. The `StockExchange` class should maintain a list of objects that implement the `StockObserver` interface and communicate its changes to the observers using the method in the `StockObserver` interface.

The `StockExchange` has the following properties:

- Stock price changes by one unit price once per second. You can use the `Thread.sleep(1000)` to make the current thread sleep for 1000 milliseconds (or make it an event-driven class using a Timer object).
- Stock price has a 50/50 probability of going up or down.
- Has the `AddObserver` and `RemoveObserver` methods that allow observers that implement the following `StockObserver` interface to add and remove themselves.

```
interface StockObserver{
      void pricedChanged(PriceChangeEvent);
}
```

You will need a `PriceChangeEvent` class that contains two fields, one for the change in price and another to the reference to the `StockExchange` object. The `StockExchange` object will communicate the change in price to the registered classes that implement the `StockObserver` interfaces (observers) through a call-back using the `priceChanged` method in the interface. These observer objects can use the information in the `PriceChangedEvent` to determine the changes in the stock price.

Create the following observer classes:

- A `StockMonitor` class that implements StockObserver interface and monitors the changes in the stock price. The `StockMonitor` should monitor and print how many steps it takes for the stock to change by a specific amount.
- A `StockPricePrinter` class that implements the `StockObserver` interface and prints the stock price after each change in the stock price.

  Since there are many common characteristics between the `StockMonitor` and `StockPricePrinter` create a class `StockCustomer` that contains the common characteristics and derive `the StockMonitor` and `StockPricePrinter` from the `StockCustomer` class.

Create a Tester class. The tester class should create an instance of `StockExchange` class and five instances of the `StockMonitor` classes, one each to monitor how many steps it takes the stock price to change by 5, 10, 15, 20, and 25 units up or down. Also, create one instance of a `StockPricePrinter` class to monitor and plot the changes in price after each change in the price.

The `StockMonitor` objects should remove themselves from the `StockExchange` when the stock prices reaches its individual threshold. The program should stop when all `StockMonitor` objects have removed themselves and print the results observed by each monitor. The `StockPricePrinter` should keep plotting the price from the beginning to the end of the run.

C. Create a program to maintain a list of students and their grades. Each student should contain the following attributes:
   a. First name
   b. Last name
   c. ID
   d. Email address
   e. List of courses taken by the students and their grades (Course description and grade).

   Class design requirements:

   Your program should contain the following classes.
   - class `StudentRecordManager`: A class that contains the list of students with the following requirements:
     - Add a student, remove a student, and search for a student by ID or last name. Search by ID should return a single student, but a search by name should check both first name and last name and return a list of student that contain the search string in their name.
     - Add a course to a student and remove a course from the student.
     - `getIterator` method returns an iterator that allows an external class to iterate through the list of students and process them. The `getIterator` method returns an iterator that implements the following interface:

```
interface <E> Iterator {
   E getNext();          //Get the next item
   boolean hasNext();    //Returns true if has more
                         //items, false otherwise.
}
```

   i. class `Student`: A class to manage the student information. Make a concise list of required attributes necessary for assignment. The `Student` class should also return an iterator that implements the `Iteraror<Course>` interface and allows you to iterate through the list of courses a student has taken.

   ii. class `Course`: An inner class to maintain the course details for each student.

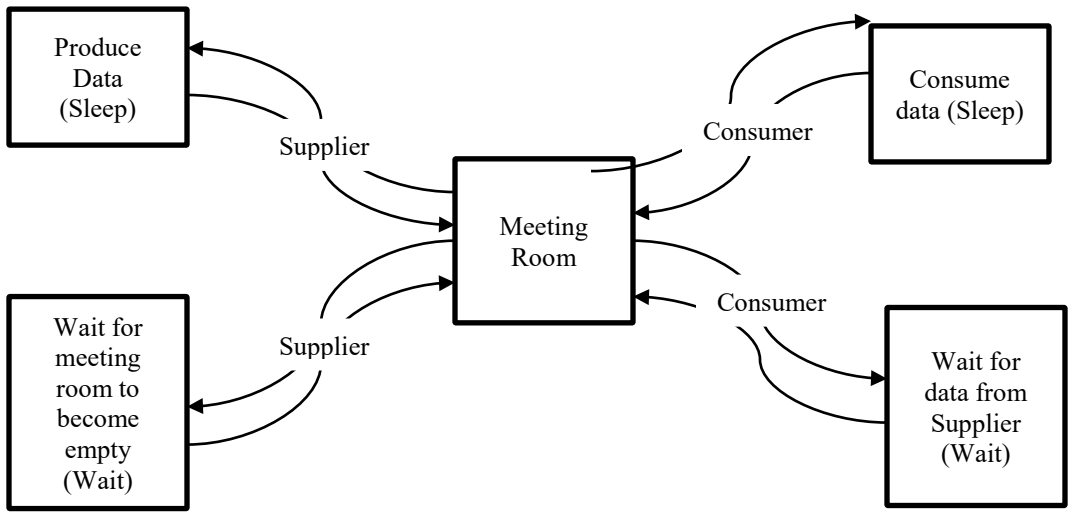
   iii. class `TranscriptPrinter`: A class that prints the transcripts. It has the following overloaded methods:

f. `printTranscript` that takes a `Student` as argument and prints the transcript for the specified student.
g. `printTranscript` that takes a `Iterator<Student>` a list of student IDs as arguments and prints the transcript for the specified students.

Use copy constructors in the Student and Course classes to prevent privacy leaks. Privacy leak occurs when you expose a private variable by returning a reference to the private variable in a accessor method.

D. Write a program to simulate the behavior of supplier and consumer threads that exchange data as described below.

Create a set of 10 supplier threads that create random data and pass the data to a set of 10 consumer threads. Each supplier and consumer thread has a unique id between 1 and 10. The supplier thread will supply the data to the consumer thread with the same matching id. The supplier and the consumer threads will meet in a synchronized common meeting room. At any time, there can be at most one supplier or consumer thread in the meeting room. Both consumer thread and supplier threads need to get access to the lock before they go in to the meeting room and leave the lock when they exit. Threads will block while competing for the lock. There can be at most one data field in the meeting room and that data will be marked for a specific consumer thread.



Their interaction is illustrated in the above figure. Each supplier will sleep for a random time up to 1000 milliseconds, wakeup, create random number, go to the meeting room, and leave the random number for its consumer thread. The consumer thread will fetch the data and sleep for a random time up to 1000 milliseconds and then go back to fetch more data. If the supplier wants to leave data, but if there is unconsumed data in the meeting room, then it will go to waiting room until it is notified. When the consumer thread goes into the meeting room and finds the data is marked for its own consumption, it will copy the data and clear the fields in the meeting room so that the next supplier can leave its data. If the data is not marked for its use, then it will go to the waiting room and wait until it is notified.

Insert diagnostic code in the supplier and consumer thread that identifies each step. For example, the supplier thread could print the following messages at each step:

```
"Supplier 0 sleeps 850 milliseconds.
 Supplier 1 sleeps 650 milliseconds.
 Supplier 0 produces data.
 Consumer 0 sleeps 950 milliseconds.
 Supplier 0 enters meeting room/leaves data for consumer 0.
 Supplier 0 leaves meeting room.
 Consumer 1 sleeps 250 milliseconds.
 Consumer 1 enters meeting room.
 Consumer 1 enters waiting room.
 Consumer 0 enters meeting room/ removes data.
 Consumer 0 sleeps for 500 millisecods.
 ……..
 ……."
```

You could also enter the times in milliseconds at each of the above stages of the thread.

E. Implement a priority queue capable of holding objects an arbitrary generic type T, by defining a generic `PriorityQueue<T extends Comparable>` class that implements the queue using an `ArrayList`. A priority queue is type of list where every item added has an associated priority and has the following methods:

   a. Add(T) – Adds a new `Comparable` item to the list.
   b. Remove() – Returns the item with the highest priority and removes it from the queue. If a user attempts to remove from an empty queue, return null.
   c. An getIterator that returns an iterator that implements the Iterator described in assignment B.

   Use the `CompareTo` method in the `Comparable` interface to compare the priority of object. You can implement the priority queue by performing a linear search through the `ArrayList`.