CPSC 210
Sample Final Exam Questions - Solution

Don't even think about looking at these solutions until you've put significant effort into developing your own solution to these problems!!!

Q1a) What does it mean for the design of a class to be robust?

First, all of the methods in the class must be robust. A method is robust if its specification covers all possible input values to that method.

In addition, we must specify class invariants and ensure that they hold before any method in the class is called and immediately after any of those methods executes. The invariants specify what an operation can assume about the state of an instance of that class at any time.

b) Design and implement a class that represents an *unchecked* exception that will be thrown by the following method of a `MyArrayList<E>` class when the index is not valid:

```
public E get(int index);
```

The class must provide a constructor that takes the invalid index as a parameter and uses it to construct a meaningful error message that can be displayed when the exception is caught.

```
/**
 * Represents an exception raised when an illegal index
 * is used.
 */
public class IllegalIndexPosition extends RuntimeException {
     /**
      * Constructor
      * @param index  the illegal index
      */
     public IllegalIndexPosition(int index) {
          super("The index " + index + " is not valid.");
     }
}
```

Q2. Suppose that a friend of yours has designed a type hierarchy where `ClassB` is a subclass of `ClassA`. `ClassB` overrides the method `doSomething` defined in `ClassA` and throws a checked exception that is not thrown by the method in `ClassA`. The code does not compile. In terms of a design principle studied this term, explain why you would not want such code to compile.

Recall that we want to be able to substitute an instance of the subclass for an instance of the super class. We would not want the code described above to compile because if we were to substitute an instance of `ClassB` for an instance of `ClassA` the method `doSomething` might throw a checked exception that the client was not expecting.

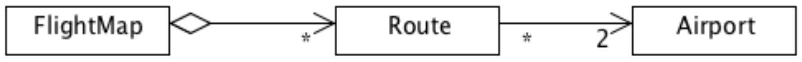
Q3. This question refers to the class `ubc.cpsc210.list.linkedList.MyLinkedList` from the `\lectures\LinkedListComplete` project.

Complete the implementation of the following member of the `linkedlist.MyLinkedList` class. Assume that `Collection<E>` is from the `java.util` package.

```
/**
 * Returns true if this list contains all the elements in the
 * collection c, false otherwise.
 */
public boolean containsAll(Collection<E> c) {
    for (E next : c) {
        if (!contains(next))
            return false;
    }

    return true;
}
```

Q4. Provide an implementation for the classes shown in the UML diagram below. You must include any fields or methods necessary to support the relationship between the classes in addition to appropriate getters and setters. Note that a route has two associated airports: the departure airport and the arrival airport. Each airport has a unique code (e.g. "YVR" represents Vancouver International, "LHR" represents London Heathrow and "PEK" represents Beijing) which cannot be changed.

Assume that once set, the arrival and departure airports for a particular route cannot be changed. Further assume that routes can be added to or removed from a flight map but the same route cannot be added to the flight map more than once. We consider two routes to be the same if they have the same departure and arrival airports. Two airports are the same if they have the same code.



```
public class Airport {
    private final String code;

    public Airport(String code) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }

     // Generate these methods using Eclipse!
    @Override
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this.getClass() != o.getClass())
            return false;

        Airport other = (Airport) o;

        return (code.equals(other.code));
    }

    @Override
    public int hashCode() {
        return code.hashCode();
    }
}
```

```
public class Route {
    private final Airport departure;
    private final Airport arrival;

    public Route(Airport dep, Airport arr) {
        departure = dep;
        arrival = arr;
    }

    public Airport getDepartureAirport() {
        return departure;
    }

    public Airport getArrivalAirport() {
        return arrival;
    }

     // Generate these methods with Eclipse!
    @Override
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this.getClass() != o.getClass())
            return false;

        Route other = (Route) o;

        return (other.arrival.equals(this.arrival)
                    && other.departure.equals(this.departure));
    }

    @Override
    public int hashCode() {
        return arrival.hashCode() * 13
                    + departure.hashCode();
    }
}


public class FlightMap {
    private Set<Route> routes;

    public FlightMap() {
        routes = new HashSet<Route>();
    }

    public void addRoute(Route r) {
        routes.add(r);
    }

    public void removeRoute(Route r) {
        routes.remove(r);
    }
}
```

Q5. For each of the scenarios below, identify which collection from the Java Collections Framework you would use and briefly justify your answer.

a) Suppose you want to simulate line-ups at a bank. There can be anywhere from one to several tellers available at any given time and each teller has their own line-up of customers. Tellers are numbered sequentially starting at position 0. You want to be able to get the line-up for a particular teller station by specifying the teller's position number. If the teller at position 2, for example, is absent, the line-up is `null`. Assume that there is a `Customer` class in the system. How would you represent the collection of line-ups?

`ArrayList<LinkedList<Customer>>`

Each line-up of customers can be represented by a `LinkedList<Customer>` which maintains entries in First-In First-Out (FIFO) order. We use a `LinkedList` because it makes it easier to remove the customer at the start of the list. Given that we have more than one line-up and that we want to have positional access to the collection of line-ups, we choose `ArrayList<LinkedList<Customer>>`. We cannot use `Set<LinkedList<Customer>>` as a `Set` does not provide positional access. When choosing a particular implementation of `List`, we go with `ArrayList` as the total number of teller stations is not likely to change often.

b) Suppose you are designing a course registration system. Assume that there is a `Student` and a `Course` class in the system. How would you store the students and courses so that you can quickly retrieve the courses in which a given student is registered?

`Map<Student, Set<Course>>` or `HashMap<Student, HashSet<Course>>`

For each student we want to be able to quickly retrieve the courses in which he/she is registered. We therefore use a map with the student as key and the courses in which the student is registered as the corresponding value. Given that a student won't register in a course more than once at any given time, we represent the collection of courses using a `Set`.

Q6. You have been asked to alter the `lectures/PhotoAlbumBase` system to make it possible for a method containing the following code to compile and execute correctly:

```
Album anAlbum = new Album("My Album");
// Put lots of photos into album
//…
for (Photo p: anAlbum) {
   // do something with each photo
}
```

For each interface, class, field or method that must be changed or added, describe the change or addition in as close to correct Java syntax as you can.

The class declaration must change:

```
public class Album implements Iterable<Photo> {
```

We must add the method iterator to `Album`…

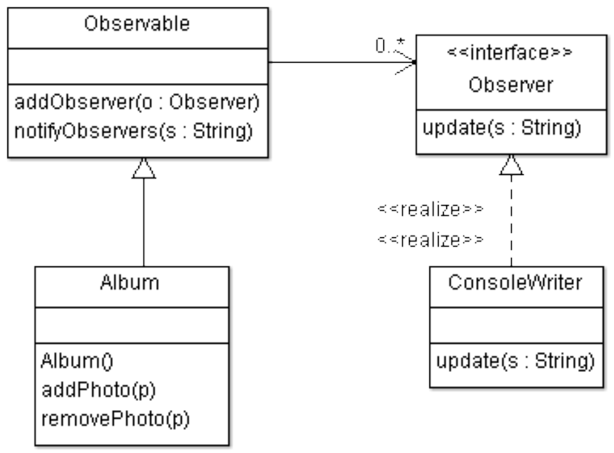
```
public Iterator<Photo> iterator() {
   return photos.iterator();
}
```

Q7. You have been given a class `ConsoleWriter` that can write a given string to the console.

```
public class ConsoleWriter {

   private writeToConsole(String s) {
      System.out.println(s); }
}
```

You have been asked to alter the functionality of the `lectures/PhotoAlbumBase` system to write to the console the name of any photo added or deleted from an album in the system. You have been told you must use the Observer design pattern to implement this new functionality.

a) Draw a UML class diagram that provides an overview of the changes and additions needed to `PhotoAlbumBase` to support the use of the Observer design pattern to provide the desired functionality. You need not reproduce the entire UML class diagram for the system. Just show those parts of the UML class diagram that must change. Indicate fields and methods that must be changed or added in the UML class diagram.



b) For each interface, class, field or method that must be changed or added, describe the change or addition in as close to correct Java syntax as you can.

1) **Implement Observable.**

```
public class Observable {

   List<Observer> observers;

   public Observable() {
     observers = new ArrayList<Observer>();
   }

   public void addObserver(Observer o) {
     observers.add(o);
   }

   public void notifyObservers(String s) {
     for (Observer o: observers)
       o.update(s);
   }
}
```

2) **Implement Observer.**

```
public interface Observer {
   public abstract void update(String s);
}
```

3) **Declaration of Album must change:**

```
public class Album extends Observable {…
```

4) **Album's constructor must change to include:**

```
public Album() {
   …
   ConsoleWriter cw = new ConsoleWriter();
   addObserver(cw);
}
```

5) **Change ConsoleWriter's declaration:**

```
public class ConsoleWriter implements Observer {…
```

6) **Must alter addPhoto(…) and removePhoto(…) to add notification to observers:**

```
public void addPhoto(Photo p) {
   …
   notifyObservers(p.getName());
}

public void removePhoto(Photo p)
   …
   notifyObserver(p.getName());
}
```

7) **Must provide implementation for update on ConsoleWriter:**

```
public void update(String s) {
   writeToConsole(s);
}
```

Q8. Consider the following Java class.

```
public class Clock {
   private Integer startTime;
   private Integer numHours;

   public Clock( Integer start, Integer hrs ) {
      startTime = start;
      numHours = hrs;
   }
}
```

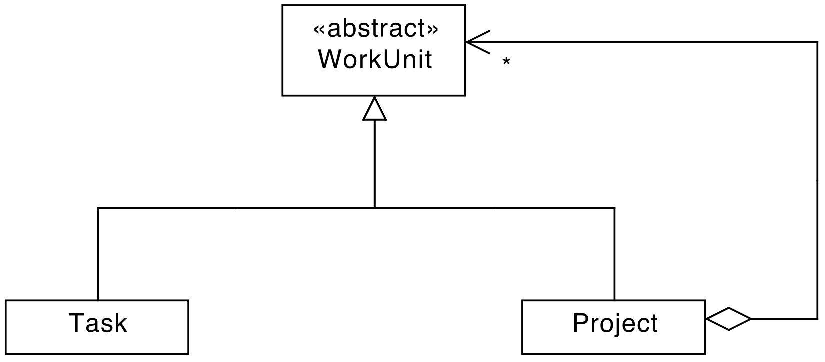
Modify this class and introduce any other code required, but without using any standard Java Collection Framework classes or interfaces (e.g., `List`), so that you can iterate over an instance of `Clock` using a for-each loop. For example, the following code should print the numbers 2, 3, 4, and 5. You do not have to consider the case where the hours go past 23. Assume the input you're given will produce output within the same day (i.e., values between 0-23 only).

```
for (Integer i: new Clock(2,4)) {
   System.out.println(i);
}
```

```
import java.util.Iterator;

public class Clock implements Iterable<Integer> {

	private Integer startTime;
	private Integer numHours;

	public Clock(Integer start, Integer hrs) {
		startTime = start;
		numHours = hrs;
	}

	@Override
	public Iterator<Integer> iterator() {
		return new ClockIterator();
	}

	private class ClockIterator implements Iterator<Integer> {

		private int currTime;

		public ClockIterator(){
			currTime = startTime;
		}

		@Override
		public boolean hasNext() {
			return currTime < (startTime + numHours);
		}

		@Override
		public Integer next() {
			Integer nextTime = currTime;
			currTime++;
			return nextTime;
		}

		@Override
		public void remove() {
			throw new UnsupportedOperationException();
		}
	}
}
```

Q9 Suppose you are designing an Android app that will allow the user to perform task management. The user can add tasks to the system and can group tasks together into projects. Projects can be added as sub-projects of other projects, nested arbitrarily deep. Each task has an estimated time for completion that is specified when the task is constructed. You want to be able to treat individual tasks and projects in the same way. In particular, you want to be able to get the time needed to complete a task or a project. The time taken to complete a project is the total time needed to complete all the tasks in the project or in sub-projects of that project.

a) Consider the partially completed code in the `TaskManager` project. Draw a UML diagram that includes all the classes in the `ca.ubc.cpsc210.taskmanager.model` package and that shows how you will apply the composite pattern to the design of this system.

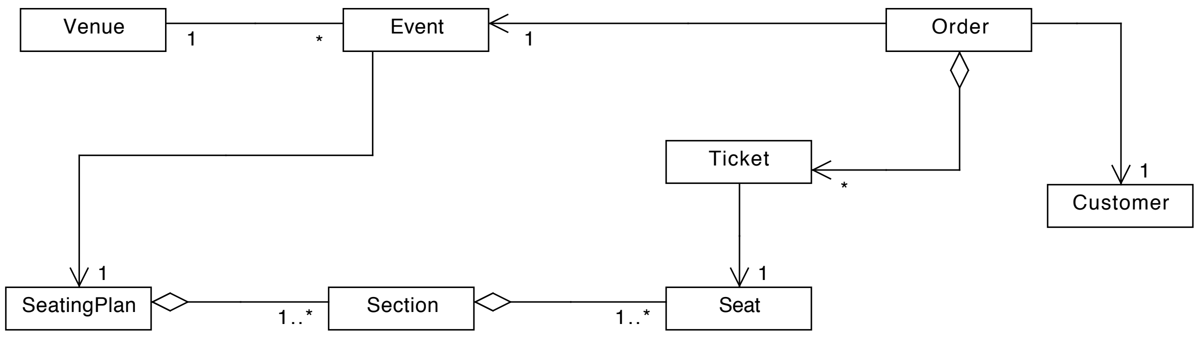


b) Write the complete implementation of the `Task` and `Project` classes below. Note that all the tests provided in `ca.ubc.cpsc210.taskmanager.tests.TestProject` should pass. Space is also provided on the following page for your answer to this question.

```
public class Task extends WorkUnit {
    private int hours;

    public Task(int hours) {
        this.hours = hours;
    }

    public int hoursToComplete() {
        return hours;
    }
}
```

```
public class Project extends WorkUnit {
    private List<WorkUnit> units;

    public Project() {
        units = new ArrayList<WorkUnit> units;
    }

    public void add(WorkUnit wu) {
        units.add(wu);
    }

    public int hoursToComplete() {
        int total = 0;

        for( WorkUnit wu : units )
            total += wu.hoursToComplete();

        return total;
    }
}
```

Q10 The following UML class diagram represents the design of a "TicketWizard" system that allows tickets to be sold for different events held at venues across the country.



Answer the following questions based on the design presented in the UML diagram above and **not** on what you think the design should be! Circle **True** or **False** and briefly explain your choice. Note that you will earn no marks if your explanation is not correct.

a) True or False: given a customer object, you can retrieve the orders placed by that customer.

False: the link between Order and Customer is uni-directional from Order to Customer.

b) True or False: it is possible that an order has no tickets associated with it.

True: the multiplicity on the Order end is *, meaning 0 or more.

c) True or False: given a ticket object, it is possible to retrieve the section object to which the associated seat belongs.

True: you can get the corresponding seat directly and the section via the bi-directional association between Seat and Section.