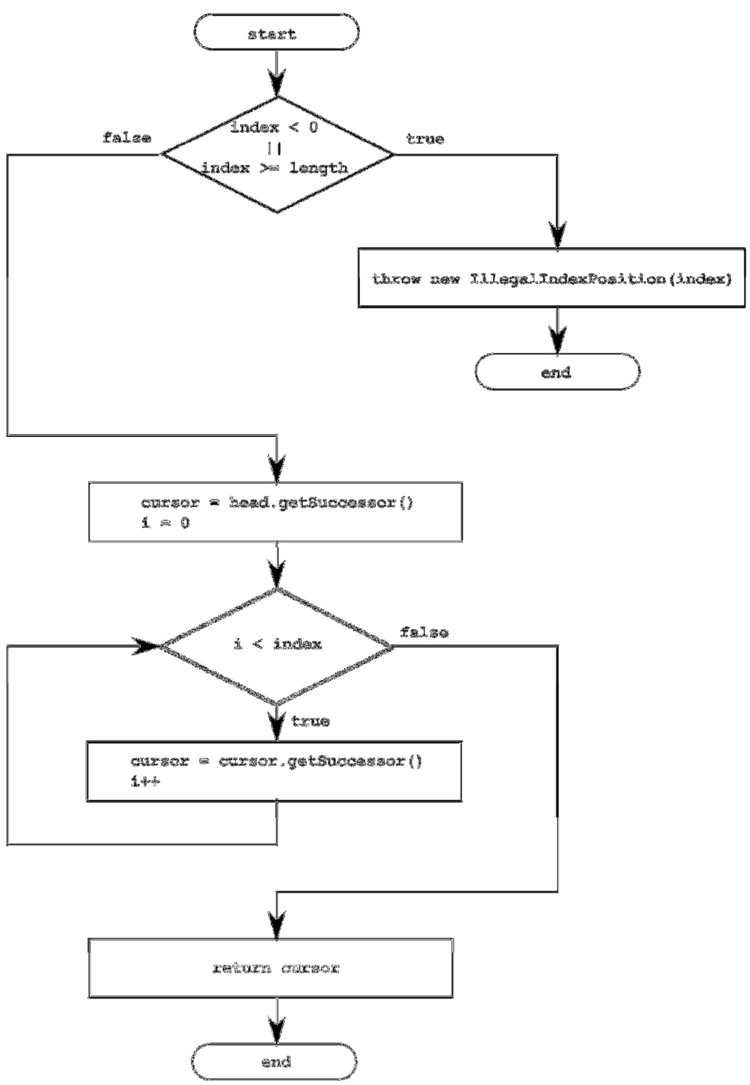
CPSC 210
Sample Final Exam Questions - Solution

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

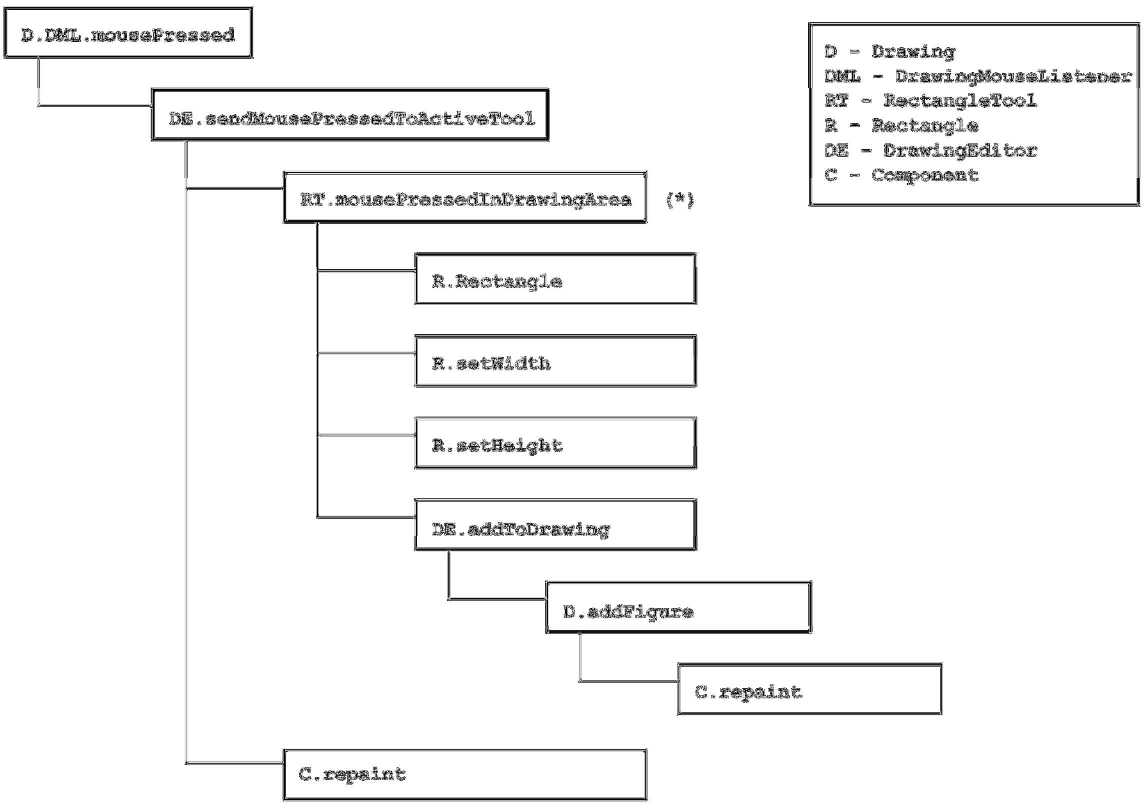
Q1. Draw an intra-method control flow diagram for the *iterative version* of the method:

```
MyListNode<E> find(int index) throws IllegalIndexPosition;
```

in the `ubc.cpsc210.list.linkedList.MyLinkedList` class of the project `\lectures\LinkedListComplete`



Q2. Draw a call graph starting at `DrawingMouseListener.mousePressed` in the `Drawing` class of the `\lectures\Decorator-Lecture-Lab-Complete` project. Assume that the active tool is a `RectangleTool` and use this information to resolve any calls to abstract methods in the `Tool` class. Include calls to methods in the `ca.ubc.cpsc210.drawingEditor.*` packages only, including methods inherited from super-types.



(*) Here's where we use the fact that the active tool is a RectangleTool

Q3a) 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 the `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.");
    }
}
```

Q4. Suppose that a friend of yours has designed a type hierarchy. At one point in the hierarchy a subtype overrides a method and throws a checked exception that is not thrown by the overridden method in the super-type. The code does not compile. In terms of a design principle studied this term, explain why you would not expect such code to compile.

In light of the Liskov Substitution Principle (LSP), we would not expect the code to compile because the overriding method in the subtype requires more of the client than the corresponding method in the super-type. If the code did compile and the client ended up having the overridden method execute, it could throw a checked exception that the client was not expecting.

Q5. This question refers to the class `ubc.cpsc210.list.linkedList.MyLinkedList` from the `\lectures\LinkedListComplete` project.

a) 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;
}
```

b) (Hard Question: This is harder than what you will find on the final but doing it will improve your knowledge of how object-oriented code works. You will have to first really understand how the code in `MyLinkedList` works.) Provide an iterative and recursive implementation of the following member of the `MyLinkedList` class:

```
/**
 *  Returns the index of the given element or -1 in the case when the
 *  element is not in the list.
 */
// Iterative version
public int indexOf(E element) {
    MyListNode<E> cursor = head.getSuccessor();
    int index = 0;

    while (cursor != tail) {
        if (cursor.getElement().equals(element))
            return index;
        index++;
        cursor = cursor.getSuccessor();
    }

    return -1;
}

// Recursive version
public int indexOf(E element) {
    return indexOfHelper(head.getSuccessor(), element, 0);
}

private int indexOfHelper(MyListNode<E> cursor, E element, int acc) {
    if (cursor == tail)
        return -1;
    else if (cursor.getElement().equals(element))
        return acc;
    else
        return indexOfHelper(cursor.getSuccessor(), element, acc + 1);
}
```

Q6. 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.


```java
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();
    }
}
```

```java
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);
    }
}
```

Q7. 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>> [or ArrayList<Queue<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. (An even better choice is to use a Queue to represent a line of customers, however we didn't cover Queues in class.)

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?

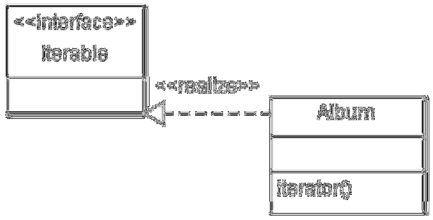
`Map<Student, Set<Course>>` or `HashMap<Student, HashSet<Course>>`

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`. We use `HashMap` and `HashSet` rather than `TreeMap` and `TreeSet` because there is no requirement that the students or courses be ordered in any particular way.

Q8. You have been asked to alter the `lectures/PhotoAlbum` 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
}
```

a) Draw the portion of the UML class diagram that provides an overview of the changes and additions needed to `PhotoAlbum` to support the code above. You need not reproduce the entire UML class diagram. 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.

The class declaration must change:

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

We must add the method iterator to Album…

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

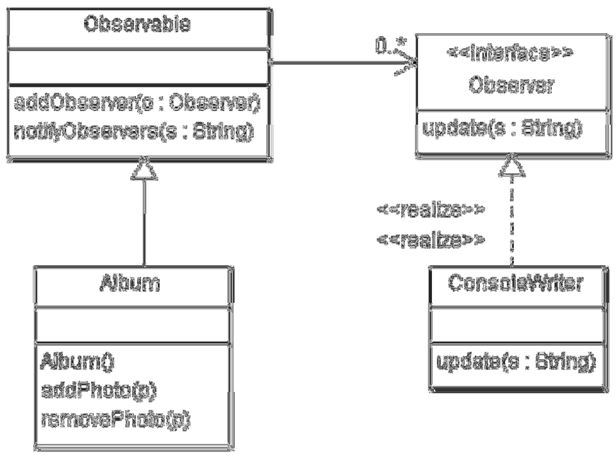
Q9. You have been given a class `ConsoleWriter` that can write a given string to the console.

```
public class ConsoleWriter {

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

You have been asked to alter the functionality of the lectures/`PhotoAlbum` 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 `PhotoAlbum` 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);
}
```

Q10. You have been asked to add new functionality to the `lectures/Pacman-refactored` code. Each time the board is redrawn (in `Board.tickBoard()`), you must write to the Java console, the number of monsters active in the system. You have been asked to design this feature by using the Observer design pattern.

Extract a design (UML Class diagram) for Pacman-refactored and alter it to show how you would support this functionality using the Observer design pattern. Include any classes or interfaces you would need to add to the design. Include any new or changed methods on new or existing classes and interfaces in the design that are needed to support the desired functionality.

Write a few sentence description of how the system would work to support this new functionality.

See packman-observer.pdf in repository for design.

In the board's constructor, the list of observers is created, a new `MonsterWriter` object is created and the `MonsterWriter` object is added to `Board` object's observers using `addObserver(...)`.

At the end of the `tickBoard()` method in `Board`, a call to `notifyObservers()` is added to let all dependent objects know the game state may have changed.

The `notifyObservers()` method calls `update(...)` with the board object as a parameter on all dependent observers. This results in the call of the overridden update method in the `MonsterWriter` class. The `MonsterWriter` uses the board parameter to call `getMonsters()` and uses the size of the returned list of monsters to write out the number of active monsters to the Java console.