Question 1 [22 marks]

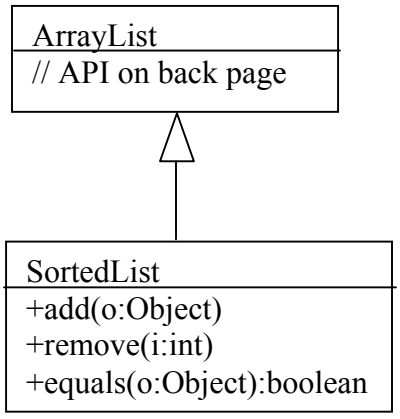
We wish to write a class that will maintain a resizable list of objects that are sorted in ascending order. We want to re-use `ArrayList`, but have to decide whether to use inheritance or composition.

- *In a real implementation, `SortedList` would implement `List` with over 15 methods. We will focus on the three methods shown in the class diagrams.*
- *You may add any variables or methods that you need.*

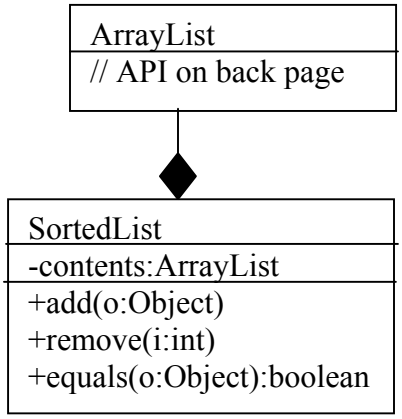
```
/** void add(Object o)
 *  Inserts the object o in the list such that the elements in the list remain in ascending
 *  order; ie. if o is stored at index i, then the object stored at i-1 is less or equal to o
 *  whereas the object stored at i+1 is greater than o.
 *  Hint: This is not a question on Iterators. Use the index to traverse the list.
 *  @throws  ClassCastException:  If the object o does not implement Comparable
 */

/** void remove(int i)
 *  Removes the element at index i
 *  @throws IndexOutOfBoundsException: If the index i is negative or > the list's size
 */
```

a) [10 marks] Write the complete implementation of the `SortedList` class **using inheritance**. For each of the three methods shown, **you must choose whether** to inherit (don't write the method) **or** override (write the method).



b) [8 marks] Now, re-write the complete implementation of the SortedList class **using composition**.



[4 marks] Is **inheritance** or **composition** a better choice for the `SortedList` ? Explain your reasons by discussing the strengths and weaknesses of the two approaches

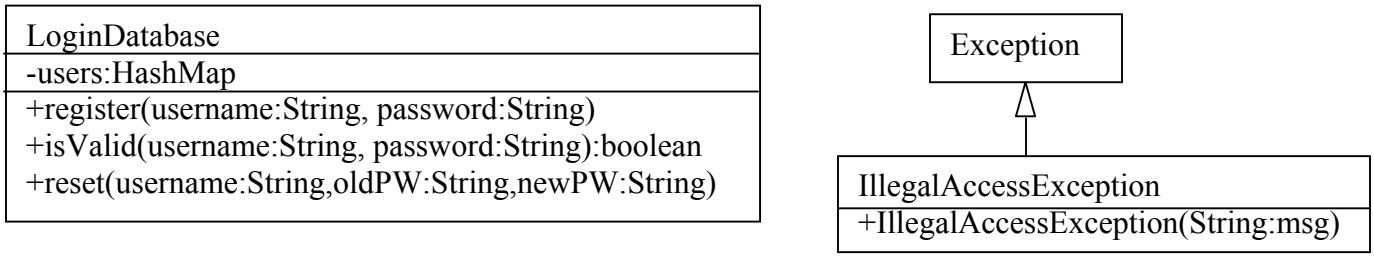
Question 2 [21 marks]

a) [3 marks] In the `HashMap` class, three methods are provided to support iteration :

```
public Set entrySet()
public Set keySet()
public Collection values()
```

Using your knowledge of `HashMap` and the `Collection` framework, explain why the first two return a `Set` while the third returns a `Collection` ?

b) [5 marks] A `LoginDatabase` uses a `HashMap` to maintain a database of {username,password} for all registered users. The key is the username, the value is the password. Below, PW is used as an abbreviation for password.



Write the implementation of the `reset`() method.

```
/** Resets the password stored for the given username to be newPW.
 * @throws IllegalAccessException if the oldPW does not match the one currently
 *                                stored for the given username or
 *                                if the username does not already exist
 */
```

c) [6 marks] A typical login dialog box is shown. The `LoginDialog` class is the implementation of this GUI. For this question, fill in the blanks in its constructor to populate the view according to the picture, and connect this class as a listener for the `ActionEvents` that will be generated when the user clicks on either of the two buttons.

The elements are laid out using a GridLayout of 3 rows, 2 columns::

JLabel	JTextField
JLabel	JTextField
JButton	JButton

```
public class LoginDialog
    extends JFrame implements ActionListener
{
 private LoginDatabase db;
 private JTextField userField, passField;

 public LoginDialog(LoginDatabase db) {
  super ( _________________);
  this.db = db;

  JPanel p = new JPanel();
  p.setLayout( ____________________________________ );

  _____.add( ____________________________________ );   //Username
  userField = new JTextField( 10 );
  _____.add( userField );

  _____.add( ____________________________________ );  //Password
  passField = new JTextField( 10 );
  _____.add( passField );

 JButton b;
  b = ____________________________________________;
  b.addActionListener( _____________ );
  _____.add( b );
  b = ____________________________________________;
  b.addActionListener( _________________);
  _____.add( b );

  this.getContentPane().add( _________________ );
 }

....  // More code in next question.
```

d) [7 marks] Now, complete the `LoginDialog's` listening code by writing the full implementation of its `actionPerformed`() method.

```
/** Accept ActionEvents from the two JButtons.
 *   If the OK button was clicked, check whether the username and password
 *   that was typed is valid. If valid, print "Accepted" and close this window
 *  ( using JFrame's dispose() method ). If not, print "Denied", and clear the two
 *  textfields so that the user can try again.
 */

public void actionPerformed( ActionEvent event)
{









  }

}
```

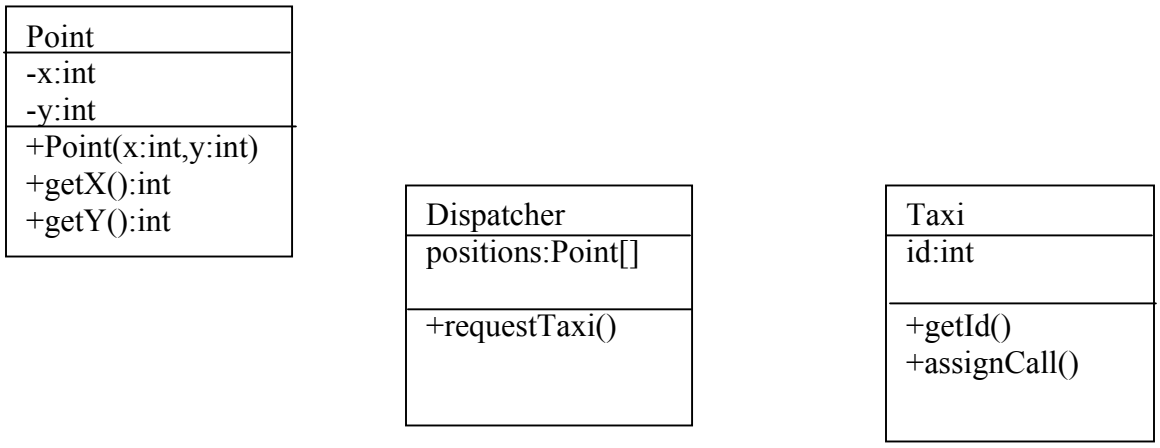
Question 3 [11 marks]

(a) [2 marks] A key feature in the Observer pattern is that the `Observable` is an abstract class. Why was an abstract class used instead of an interface ?

(b) [1 mark] What is the disadvantage of having `Observable` as an abstract class ?

(c) [4 marks] The Carleton Taxi company has all of its taxis GPS-enabled (GPS=Global Positioning System). Whenever a taxi moves, its GPS transmits the new position data for the taxi (as a Point object). The company's central dispatcher monitors all of these position updates so that it can efficiently assign incoming calls to the taxi closest to the call. Because this company has a pre-known fixed number of *n* taxis, the position updates are stored in a simple array of *n* `Point` objects (x,y) where the array's index correspond to the `Taxi's` id.

Apply the Observer pattern to this system. **Complete the UML class diagram** showing all class relationships and significant methods and/or attributes.



(d) [4 marks] Write the observer's `update()` method. Consult the API on the back page.

Question 4 [11 marks]

The following code represents a simple `PlaneTicket` class, which is mainly used to calculate the final price of the ticket based on factors such as the type of ticket and the lateness of the booking: (Question is on next page)

```
import java.util.*;

public class PlaneTicket {
	private static int ECONOMY = 0;
	private static int BUSINESS = 1;
	private static int FIRST_CLASS = 2;

	private int type;
	private Date flightDate;
	private Date dateOfIssue;
	private double travellingDistance;

	public PlaneTicket(int type, Date flightDate, double distance){
		this.type = type;
		this.flightDate = flightDate;
		this.travellingDistance = distance;
		this.dateOfIssue = new Date(); //today!
	}

	public double calculateFinalPrice() {
		double baseFare = 0;
		if ((type == BUSINESS) || (type == FIRST_CLASS)) {
			baseFare = travellingDistance * 1.5; // $1.5/km  luxury rate
		}
		else if (type == ECONOMY) {
			baseFare = travellingDistance; // $1/km economy base rate
		}
		//now add late booking penalty (less than 10 days before flight)
		// only applies to economy and business
		if ((type == ECONOMY) || (type == BUSINESS)) {
			Date d = (Date) dateOfIssue.clone();
			d.setDate(d.getDate() + 10); //10 days from date of issue
			if (flightDate.before(d)) return baseFare * 1.2;
		}
		return baseFare;
	}

}
```

a) [7 marks] The problem with the code is that, as more ticket types are introduced, the logic of the calculation of the price gets harder and harder to maintain. **Refactor** the above **design** to clean-up the calculation of the final price by eliminating the `type` information in favor of polymorphism **and** to force everyone who introduces a new type of ticket to provide a calculation of its price. **Draw the resulting UML class diagram**. (No code should be written in this question)

b) [4 marks] Write all the code pertaining to a first class ticket in your solution of Part a).

Java API – Classes and methods required on this exam.

class java.lang.ArrayIndexOutOfBoundsException extends RuntimeException
class java.lang.ClassCastException extends RuntimeException
interface java.lang.Comparable
public int compareTo(Object o) // Throws ClassCastException if o is not an instance
// of this class. Returns positive for greater, negative for less, zero for equal

class javax.swing.JFrame extends JComponent
public JFrame(String title) public void dispose() // Close the window.
class javax.swing.JButton extends JComponent
public JButton(String text) // Constructs a button with given text
public void addActionListener(ActionListener l)
class javax.swing.JTextField extends JComponent
public JTextField(int size) // Constructs of textfield of maximum size
public void setText(String text); public String getText();
public void addActionListener(ActionListener l)

interface java.awt.event.ActionListener
public void actionPerformed(ActionEvent e);
class java.util.EventObject
public Object getSource() // Return the object on which the event initially occurred.
class java.awt.event.ActionEvent extends EventObject
public ActionEvent(Object source, int id, String command)
public String getActionCommand() // Returns command associated with this action.

interface java.util.Collection
Iterator iterator(); // Returns an iterator over the elements in this collection
interface java.util.Iterator
boolean hasNext(); // Returns true if the iteration has more elements.
Object next(); // Returns the next element in the iteration.

abstract class java.util.Observable
public void addObserver(Observer o) public void deleteObserver(Observer o)
public void notifyObservers(Object arg) public boolean hasChanged()
protected void setChanged() protected void clearChanged()
interface java.util.Observer
void update(Observable o, Object arg)

class java.util.HashMap extends AbstractMap implements Map, Cloneable
public HashMap() public Object remove(Object key)
public boolean containsValue(Object value) public boolean containsKey(Object key)
public Object get(Object key) public Object put(Object key, Object value)
public Set keySet() // Returns the set of all keys in table
public Collection values() // Returns the collection of values contained in table.
public Set entrySet() // Returns all <key,value> entries in table.

class java.util.ArrayList extends AbstractList implements List, Cloneable
public ArrayList() public int size()
public boolean contains(Object value) public Object get(int index) **
public boolean add(Object o) public void add(int i, Object o) **
public Object remove(int index) **
** All methods that have an index parameter throw IndexOutOfBoundsException if the index is negative or larger than the list's size.