```
import java.io.Serializable;
import java.util.List;
import java.util.Vector;

/**
 * Lista di elementi omogenei di tipo Studenti. E' modificabile e ordinata
 * secondo i criteri di ordinamento stabiliti per gli oggetti di tipo Studenti.
 * Gli elementi della lista differiscono fra loro per nome, cognome e matricola.
 *
 * L'oggetto e' serializabile.
 *
 * @author Filippo Fontanelli , Francesca Brogi
 */
public class RegStudenti implements Serializable {

    // ************************************************

    private static final long serialVersionUID = 2L;

    // Rappresentazione

    /**
     * Collezione di oggetti di tipo Lezioni. Lista di Lezioni ordinate per
     * Cognome, Nome e Matricola.
     */
    private List<Studente> list;

    // Costruttori

    /**
     * Costruttore.
     *
     * Inizializza <code>list</code> alla Lista Studenti vuota.
     */
    public RegStudenti() {
        list = new Vector<Studente>();
    }

    /**
     *
     * Costruttore che i nizializza <code>list</code> alla Lista Studenti.
     *
     * @param list
     *             List<Studente> rappresentante la lista degli Studenti.
     *
     * @exception <code>NullPointerException</code> se la lista &egrave;
     *             <code>null</code>.
     *
     */
    public RegStudenti(List<Studente> list) {
        if (list == null) {
            throw new NullPointerException("RegLezioni::Costruttore::null");
        }
```

```
        this.list = list;
    }

    // Visitatori

    /**
     * Restituisce il numero di <Studenti> presenti in <code>list</code>.
     *
     * @return numero di elementi di list.
     */
    public int size() {
        return list.size();
    }

    /**
     * Restituisce la <code>list</code> di <Studente>.
     *
     * @return List contentr la lista degli Studenti.
     */
    public List<Studente> getElemReg() {
        return list;
    }

    /**
     * Restituisce la rappresentazione testuale della ListaFoto.
     *
     * @return String contentente le informazioni su tutti gli Studenti di
     *         <code>list</code>.
     */
    public String toString() {
        String s = "";

        for (int i = 0; i < list.size(); i++)
            s += list.get(i).toString() + "\n";

        return s;
    }

    /**
     * E' un algoritmo di ricerca binaria sulla collezione <code>list</code>.
     * Stabilisce se <code>s</code> e' presente nella collezione, basandosi sui
     * criteri di uguaglianza degli oggetti di tipo Studenti. (Due oggetti
     * Studenti sono uguali quando hanno identico Cognome, Nome e Matricola).
     *
     * @param s
     *            oggetto di tipo Studnete. Dobbiamo stabilirne la presenza
     *            nella collezione.
     *
     * @return <code>true</code> se s e' presente in list, <code>false</code>
     *         altrimenti.
     */
    public boolean isIn(Studente s) {
        if (s == null)
```

```
            throw new IllegalArgumentException("RegLezioni::isIn");
        if (list.size() == 0)
            return false;

        if (list.contains(s))
            return true;
        else
            return false;
    }

    // Modificatori

    /**
     * Aggiunge una Studente passato come parametro alla collezione. Se
     * <code>s</code> e' null, solleva NullPointerException. Se <code>s</code>e'
     * gia' presente nella collezione, solleva DuplicatedLezioniException.
     * Altrimenti inserisce <code>s</code> in <code>list</code> mantenendo
     * l'ordinamento.
     *
     * @param s
     *            oggetto di tipo Studente che deve essere inserito nella
     *            collezione di dati.
     *
     * @exception <code>DuplicatedStudentiException</code> se nell'archivio e'
     *            presente un duplicato dello Studente che stiamo tentando di
     *            inserire.
     *
     * @exception <code>NullPointerException</code> se lo Studente passato come
     *            parametro e' null.
     */
    public void addStudenti(Studente s) throws DuplicatedStudentiException,
            NullPointerException {

        if (s == null)
            throw new NullPointerException("RegLezioni::addStudenti");
        // di seguito si fa il controllo sull'eventuale presenza di un duplicato
        if (list.indexOf((Studente) s) >= 0)
            throw new DuplicatedStudentiException("RegLezioni::addStudenti");

        if (list.size() == 0) {
            list.add(s);
        } else {
            if (list.get(size() - 1).compareTo(s) < 0) {
                list.add(size(), s);
            } else {
                for (int i = 0; i < list.size(); i++)
                    if (list.get(i).compareTo(s) > 0) {
                        list.add(i, s);
                        return; // una volta inserito l'elemento, si esce dal
                        // ciclo.
                    }
            }
        }
```

```
}

/**
 * Rimuove lo Studente passato come parametro alla collezione. Se
 * <code>s</code> e' null, solleva NullPointerException. Se <code>s</code>
 * non e' presente nella collezione, solleva IllegalArgumentException.
 * Altrimenti elimina <code>s</code> da <code>list</code> mantenendo
 * l'ordinamento.
 *
 * @param s
 *            oggetto di tipo Studenti che deve essere inserito nella
 *            collezione di dati.
 *
 * @exception <code>IllegalArgumentException</code> se nell'archivio non e'
 *            presente lo Studente che stiamo tentando di rimuovere.
 *
 * @exception <code>NullPointerException</code> se lo Studente passato come
 *            parametro e' null.
 */
public void removeStudenti(Studente s) throws NotExistentStudentiException,
        NullPointerException {

    int indexS;
    if (s == null)
        throw new NullPointerException("RegLezioni::removeStudenti");
    // di seguito si fa il controllo sull'eventuale presenza di un duplicato

    if (list.indexOf((Studente) s) < 0)
        throw new NotExistentStudentiException("RegLezioni::removeStudenti");
    // ottengo l'indice dello studente e lo elimino
    indexS = list.indexOf((Studente) s);

    list.remove(indexS);

}

/**
 * Modifica lo Studente relativo all'indice <code>index</code> passato come
 * parametro.
 *
 * @param index
 *            indice di riferimento dello Studente nella collezione.
 *
 * @exception <code>IllegalArgumentException</code> se nell'archivio non e'
 *            presente lo Studente che stiamo tentando di rimuovere.
 *
 * @exception <code>NullPointerException</code> se lo Studente passato come
 *            parametro e' null.
 */
public void modify(int index) {

    System.out.println("Dettagli sullo studente :");
    System.out.println(this.list.get(index).toString());
```

```
        System.out
                .println("Inserire il cognome ( * per lasciarlo invariato) :\n");
        String cognome = SafeInput.readLineSafe();
        if (!cognome.equals("*")) {
            this.list.get(index).setCognome(cognome);
        }

        System.out.println("Inserire il nome ( * per lasciarlo invariato) :\n");
        String nome = SafeInput.readLineSafe();
        if (!cognome.equals("*")) {
            this.list.get(index).setNome(nome);
        }

        int mat = 0;

        System.out
                .println("Inserire la matricola (0 per lasciarlo invariato) :\n");
        mat = SafeInput.readInt();

        if (mat != 0) {
            this.list.get(index).setMatricola(mat);
        }
        return;
    }

    /**
     * Cerca lo Studente relativo alla Matricola <code>mat</code> passata come
     * parametro.
     *
     * @param mat
     *            matricola di riferimento dello Studente nella collezione.
     *
     * @return int rappresentante l'indice di riferiemnto dello Studente con
     *         matricola uguale a mat.
     */
    public int cercaMat(int mat) {
        boolean trovato = false;
        int i = 0;
        while (trovato == false && i < list.size()) {
            if (this.list.get(i).getMatricola() == mat) {
                trovato = true;
            } else
                i++;
        }
        if (i == this.list.size()) {
            return -1;
        } else
            return i;
    }
}
```