Fondamenti di informatica T-1 (A – K)
Esercitazione 10

Ereditarietà, interfaccia *Comparable*, collezioni

AA 2018/2019

Tutor

Lorenzo Rosa

lorenzo.rosa@unibo.it

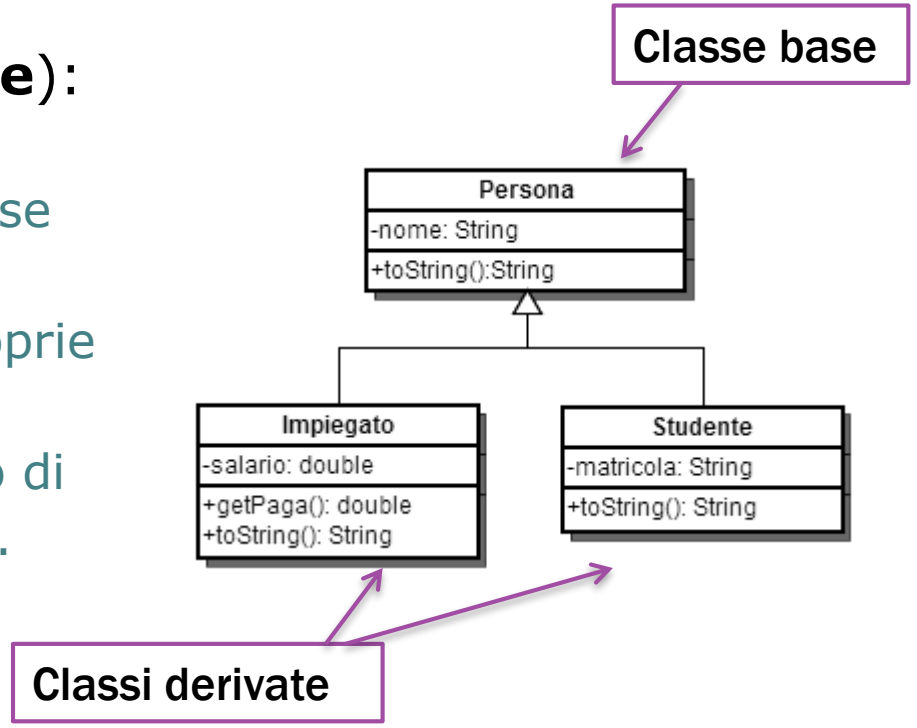
Esercitazione 10

- Introduzione al calcolatore e Java
- Linguaggio Java, basi e controllo del flusso
- I metodi: concetti di base
- Stringhe ed array
- Classi e oggetti, costruttori, metodi statici, visibilità
- Eclipse, ereditarietà e polimorfismo
- Collezioni Java
- Esercizi d'esame

Ereditarietà

- Meccanismo per definire una nuova classe (classe derivata) come specializzazione di un'altra (classe base)
  - La classe base modella un concetto generico
  - La classe derivata modella un concetto più specifico

- La classe derivata (**sottoclasse**):
  - Dispone di tutte le funzionalità (attributi e metodi) di quella base
  - Può aggiungere funzionalità proprie
  - Può ridefinirne il funzionamento di metodi esistenti (polimorfismo).



Ereditarietà

- Processo di astrazione
  - Si introduce la superclasse che "astrae" il concetto comune condiviso dalle diverse sottoclassi
  - Le sottoclassi vengono "spogliate" delle funzionalità comuni che migrano nella superclasse
- Ogni classe definisce un tipo:
  - Un oggetto, istanza di una sottoclasse, è formalmente compatibile con il tipo della classe base
  - Il contrario non è vero!
- Ad esempio
  - Un impiegato è una persona ma una persona non è (necessariamente) un impiegato
  - Un'automobile è un veicolo ma un veicolo non è (necessariamente) un'automobile

Esempio

```java
public class Persona {

    private String nome;
    public Persona(String nome) { this.nome = nome; }
    public String toString()    {return "Mi chiamo " + this.nome;}
}
```

```java
public class Studente extends Persona{
    private String matricola;
    public Studente(String nome, String matricola) {
        super(nome);
        this.matricola = matricola;
    }
    @Override
    public String toString() {
        return super.toString() +
        ", numero di matricola: " + this.matricola;
    }
}
```

```java
public class Impiegato extends Persona {
    private double salario;
    public Impiegato(String nome, double salario) {
        super(nome);
        this.salario = salario;
    }
    public double getPaga() { return salario;}
    @Override
    public String toString() {
        return super.toString() + " e guadagno " + getPaga() + "€"; }
}
```

Terminologia

Parola chiave ***"extends"*** :
Specifica da quale classe eredita.
Nell'esempio, Studente eredita da Persona

Parola chiave ***"super"*** :
Consente di invocare un metodo, un costruttore o un attributo della classe base purché non privati

Annotazione ***"@Override"*** :
Permette di ridefinire un metodo della superclasse a condizione che abbia stesso nome, parametri e tipo di ritorno (magari stessa semantica)

```
public class Studente extends Persona{
    private String matricola;
    public Studente(String nome, String matricola) {
        super(nome);
        this.matricola = matricola;
    }
    @Override
    public String toString() {
        return super.toString() +
        ", numero di matricola: " + this.matricola;
    }
}
```

Il metodo *toString*

Ogni classe eredita da `Object` il metodo:

```
public String toString()
```

che restituisce una stringa che difficilmente è quella che desideriamo.

Esempio:

```
Libro p = new Libro("Giuseppe");
System.out.println(p);
```

La stringa stampata in questo caso è:

Libro@15db9742

... non proprio leggibile!

Il metodo *toString*

Per avere una stampa significativa, dobbiamo ridefinire il metodo `toString()` della nostra classe.

```
public class Libro {
    ...
    public String toString() {
        return this.titolo + … ;
    }
}
```

Esempio

```
Libro p = new Libro("Che bello Java");
System.out.println(p);
```

La stringa stampata in questo caso è:

```
Che bello Java
```

... molto meglio!

Il metodo *equals*

Ogni classe eredita da `Object` anche il metodo:

```
public boolean equals(Object o)
```

che indica se "un oggetto *o* è uguale a quello corrente". Così come le stringhe, anche gli altri oggetti si confrontano in questo modo (e non con "==").

```
Libro p1 = new Libro("Che bello Java");
Libro p2 = new Libro("Che bello Java");
bool res = p1.equals(p2); //true
```

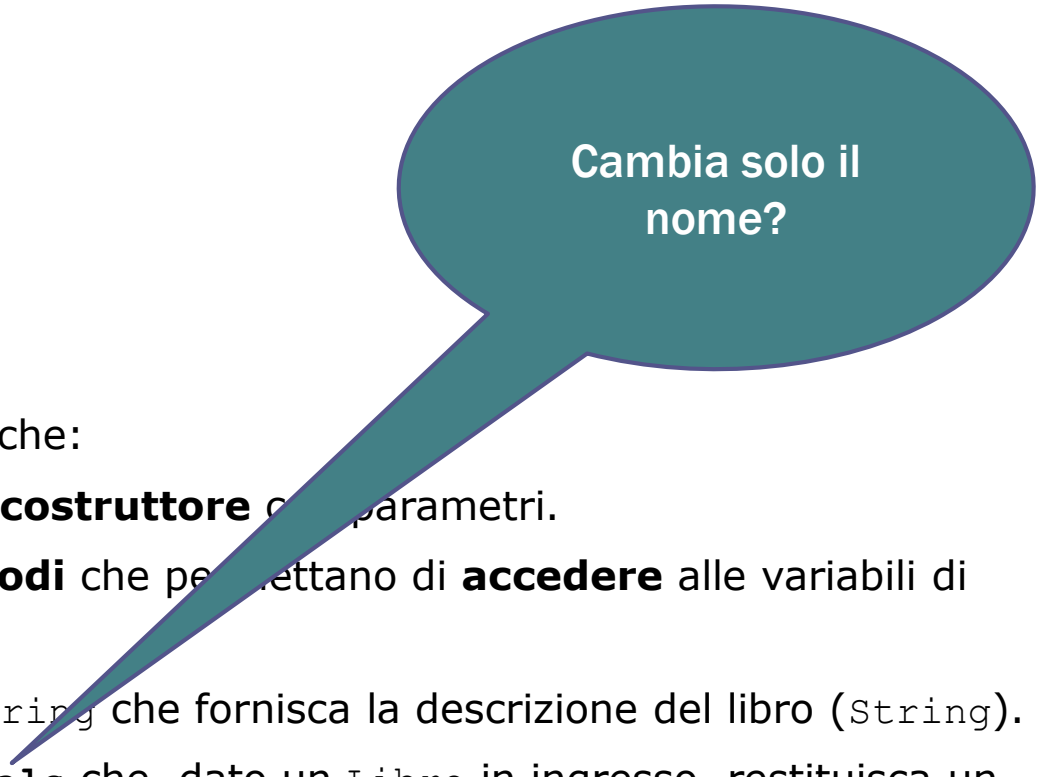
Problema:

il metodo *equals* della classe Persona prende in ingresso **un generico oggetto *o***, che potrebbe non essere una persona!

Libro (uguale a esercitazione 8)

Il rettore ha deciso di informatizzare la gestione dei libri dell'ateneo. Per ogni libro occorre memorizzare:

1. l'autore
2. il titolo
3. la casa editrice
4. l'anno di pubblicazione.



Si scriva una classe `Libro` che:

1. Possieda un opportuno **costruttore** con parametri.
2. Presenti opportuni **metodi** che permettano di **accedere** alle variabili di istanza dell'oggetto.
3. Presenti il metodo `toString` che fornisca la descrizione del libro (`String`).
4. Presenti il **metodo** **`equals`** che, dato un `Libro` in ingresso, restituisca un booleano: `true` se tutti i campi del Libro in ingresso sono uguali ai propri, `false` altrimenti.

Il metodo *equals* di Libro

- Il vecchio metodo *uguale:*

```
public boolean uguale(Libro altro) { … }
```

- Il nuovo metodo *equals:*

```
public boolean equals(Libro altro) { … }
```

- Il metodo ereditato da Object (che siamo **obbligati** a usare):

```
public boolean equals(Object altro) { … }
```

Il metodo *equals* di Libro

- Il metodo ereditato prende in ingresso un Object:

```
public boolean equals(Object altro) { … }
```

ovvero potenzialmente qualunque cosa, non solo un Libro.

- Quindi? Il ragionamento è il seguente:
  - Se "altro" **non è** una istanza di Libro, sicuramente non è uguale al libro corrente (può essere un Contatore uguale a un Libro?)
  - Se "altro" è un'istanza di Libro, allora dobbiamo:
    1. **fare un cast a Libro (altrimenti lo trattiamo come un oggetto generico).**
    2. **invocare la funzione che abbiamo già scritto.**

- Come si traduce in codice?

Il metodo *equals*

1. Se "altro" **non è** una istanza di `Libro`, sicuramente non è uguale al libro corrente (può essere un `Contatore` uguale a un `Libro`?)

```
public boolean equals(Object altro) {
  if( altro instanceof Libro )
       // punto 2
   else return false;
}
```

Se "altro" non è un Libro, certamente non è uguale al libro corrente!

instanceof è una keyword Java che serve a controllare che una variabile contenga un oggetto della classe specificata di seguitp. Nell'*if* controllo che obj sia davvero di classe *Persona,* così posso effettuare il *cast* senza problemi.

Il metodo *equals*

2. Se "altro" è un'istanza di Libro, allora dobbiamo:
   1. fare un cast a Libro (altrimenti lo trattiamo come un oggetto generico).
   2. invocare la funzione che abbiamo già scritto.

```
public boolean equals(Object altro) {
  if( altro instanceof Libro )
    return this.equals((Libro)altro);
  else return false;
}
```

Cast a Libro

Il metodo *equals* che abbiamo già scritto!

Il metodo *equals*

La soluzione completa del punto 4:

```
public boolean equals(Object altro) {
       if( altro instanceof Libro )
              return this.equals((Libro)altro);
       else return false;
}

public boolean equals(Libro altro) {
       return this.autore.equals(altro.getAutore())
       && this.titolo.equals(altro.getTitolo())
       && this.casaEditrice.contentEquals(altro.getCasaEditrice())
       && this.annoPubblicazione == altro.getAnnoPubblicazione();
}
```

Oltre la equals

- Perché non ci bastava il metodo *equals(Libro)?* Perché in alcune situazioni (vedi fine esercitazione), il metodo *equals* viene chiamato in automatico da Java. La versione che viene chiamata è quella ereditata: *equals(Object).*

- Grazie a *equals* otteniamo così la possibilità di esprimere l'uguaglianza tra oggetti di classi scritte da noi.

- Se vogliamo anche dire se un'istanza di Libro è maggiore o minore di un'altra? Non bastano i metodi che possiamo ereditare, ma dobbiamo **implementare un'interfaccia.**

Interfaccia *Comparable<T>*

```
public interface Comparable<T> {
        int compareTo(T o);
}
```

- Quest'interfaccia serve per definire dei criteri di ordinamento tra un oggetto ed un altro.

- Contiene al suo interno un solo metodo da ridefinire.

compareTo

Il metodo *compareTo* viene usato per confrontare oggetti secondo **l'*ordinamento naturale***, che segue la relazione:

$$\{(x, y) \mid x.compareTo(y) <= 0\}.$$

Sta a noi, che implementiamo *compareTo*, restituire un valore che, in base ai nostri criteri (crescente o decrescente), consenta di ordinare oggetti secondo questa relazione.

compareTo: esempio

Supponiamo di avere la classe:

```
public class Venditore implements Comparable<Venditore>
{
	private int frigoriferiVenduti;

	...

	public int compareTo(Venditore v) {
		// cosa restituisco?
	}
}
```

e di volere definire l'ordinamento **in base ai frigoriferi venduti.**

compareTo

Cosa significa definire l'ordinamento in base ai frigoriferi venduti? Supponiamo di avere due venditori:

```
Venditore v1, v2;
```

allora il metodo

```
v1.compareTo(v2);
```

deve restituire un intero negativo, zero o un intero positivo a seconda che (il numero di frig. venduti da) **v1 sia minore, uguale o maggiore a** (il numero di frig. venduti da) **v2**.

compareTo: ordine crescente

ORDINE CRESCENTE: devo mettere prima i valori più piccoli quindi, nel nostro caso, prima i venditori con meno frigoriferi.

Ovvero, secondo la semantica dell'ordinamento naturale, v1.compareTo(v2) deve restituire un valore:

negativo se v1 ha venduto meno frigoriferi di v2

0 se v1 ha venduto lo stesso numero di frigoriferi di v2

positivo se v1 ha venduto più frigoriferi di v2

compareTo: ordine crescente

```
public class Venditore {

  ...

 public int compareTo(Venditore v2) {

    return this.frigoriferiVenduti - v2.frigoriferiVenduti;

  }

}
```

compareTo: ordine decrescente

ORDINE DECRESCENTE: devo mettere prima i valori più grandi, quindi prima i venditori con più frigoriferi.

Ovvero, secondo la semantica dell'ordinamento naturale, v1.compareTo(v2) deve restituire:

negativo se v1 ha venduto più frigoriferi di v2

è più grande, quindi andrà messo **prima** dell'altro!

0 se v1 ha venduto lo stesso numero di frigoriferi di v2

positivo se v1 ha venduto meno frigoriferi di v2

è più piccolo, quindi andrà messo **dopo** l'altro!

compareTo: ordine decrescente

```
public class Venditore {

  ...

 public int compareTo(Venditore v2) {

    return v2.frigoriferiVenduti - this.frigoriferiVenduti;

 }

}
```

Libro (versione originale)

Il rettore ha deciso di informatizzare la gestione dei libri dell'ateneo. Per ogni libro occorre memorizzare: l'autore, il titolo, la casa editrice, l'anno di pubblicazione.

Si scriva una classe `Libro` che:

1. Possieda un opportuno costruttore con parametri.
2. Presenti opportuni metodi che permettano di accedere alle variabili di istanza dell'oggetto.
3. Presenti il metodo `toString` che fornisca la descrizione del libro (`String`).
4. Presenti il metodo `equals` che, dato un `Libro` in ingresso, restituisca un booleano: `true` se tutti i campi del Libro in ingresso sono uguali ai propri, `false` altrimenti.
5. Implementi l'interfaccia Comparable, definendo il metodo **compareTo** per stabilire la precedenza con un Libro passato come parametro (per autore ed anno di pubblicazione crescenti).

Libro (versione originale)

```
class Libro implements Comparable<Libro> {
  …
  public int compareTo(Libro li) {
        int ret = this.autore.compareTo(li.autore);
        if(ret == 0)
           ret = this.anno - li.anno;
         return ret;
     }
}
```

Collections

- Nella scorsa esercitazione abbiamo visto gli *array.* Ci risolvevano molti problemi, ma:
  - Hanno una dimensione fissa: siamo costretti a usare un parametro nel costruttore per indicare la dimensione massima;
  - Occupano memoria per un numero fisso di elementi, anche se ne usiamo solo una piccola parte;
  - Per aggiungere un elemento dobbiamo "ricordarci" a che indice abbiamo inserito l'ultimo (dimensione logica).

- Tutti questi svantaggi si possono evitare grazie all'uso di Collection

Collections

`java.util.Collection` è **un'interfaccia**, radice di una tassonomia di interfacce e di classi concrete.

Una C*ollection* rappresenta una qualunque collezione di oggetti:

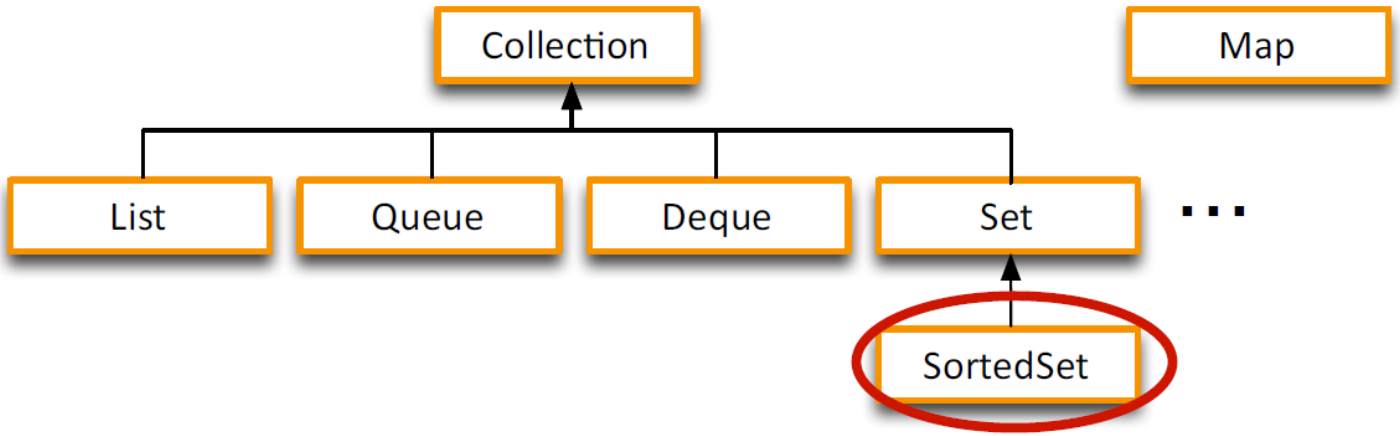
- Alcune collezioni permettono l'esistenza di duplicati, altre no.
- Alcune sono collezioni ordinate, altre no. L'ordinamento viene effettuato secondo il criterio definito nella *compareTo(Object)*

La **JDK** non prevede implementazione alcuna per questa interfaccia: prevede invece classi (come *TreeSet* ed ***ArrayList***) che implementano interfacce più specifiche da essa derivate (come Set e List).

Collection

L'interfaccia *Collection* definisce (ma non implementa) alcuni metodi base, quali :

- `boolean add (Element e)` per inserire un elemento
- `boolean remove (Element e)` per eliminare un elemento
- `int size()` restituisce il numero di elementi nella collezione
- `Bollean isEmpty()` restituisce true se è vuota



Esempio pratico: ArrayList

ArrayList<T> rappresenta una collezione in cui:

- posso inserire l'emento i-esimo ad un indice specifico
  - `add(T e)`

    Inserisce l'elemento *e* **in coda** alla lista.
  - `add(int index, T element)`

    Inserisce l'elemento *e* in posizione *index* nella lista.
- Possono apparire duplicati.
- La classe ArrayList implementa l'interfaccia List.

Esempio pratico: ArrayList<T>

Esempio: una lista di stringhe

```
List<String> strings = new ArrayList<String>();

strings.add("Two");
strings.add("Three");
strings.add(0, "One");
strings.add(3, "One");
strings.add("Three");
strings.add(strings.size() - 1, "Two");

System.out.println(strings);
// Output: [One, Two, Three, One, Two, Three]
```

Array VS Arraylist<T>

Array	ArrayList<T>
Contengono solo oggetti dello stesso tipo, non possiamo avere in uno stesso Array una Stringa ed un intero.	Contengono solo oggetti dello stesso tipo T, non possiamo avere in uno stesso Array una Stringa ed un intero.
Dimensione **prefissata**, non estendibile.	Dimensione iniziale **non prefissata**: posso aggiungere o rimuovere elementi senza occuparmi della dimensione fisica, che viene scalata automaticamente.

Esempio pratico di set: HashSet<T>

`HashSet<T>` rappresenta una collezione che:

- **non garantisce ordinamento**:
  - in un insieme non c'è la nozione di ordinamento.

- **non consente duplicati**: non può esistere una coppia di oggetti di tipo T, e1 ed e2, tali per cui e1.equals(e2).
  - La add() restituisce un **valore boolean**: se è falso, significa che l'elemento è già presente nell'insieme e quindi non è stato aggiunto.

- La classe HashSet<T> implementa l'interfaccia l'interfaccia **Set<T>**

Esempio pratico di set: HashSet

Esempio d'utilizzo:

```
Set<String> strings = new HashSet<String>();
strings.add("One");   // true
strings.add("Two");   // true
strings.add("One");   // false: c'è gia!
strings.add("Three"); //true

System.out.println(strings.toString());
```

Output: [One, Three, Two]

Insieme ordinato: SortedSet<T>

`SortedSet<T>` è una interfaccia che implementa `Set<T>` e che rappresenta un **insieme ordinato**, quindi una collezione che impone **ordinamento naturale** (serve che T implementi l'interfaccia *Comparable*):

```
add(T el)
```

aggiunge l'elemento "*el*", se non è già presente (altrimenti, restituisce false), collocandolo in ordine rispetto agli elementi già presente.

Non ammette duplicati, cioè non può esistere una coppia di elementi *e1* ed *e2* tali per cui `e1.equals(e2)`.

Esempio pratico: TreeSet<T>

- `SortedSet<T>` è un' interfaccia.
- La classe concreta che useremo come implementazione di `SortedSet<T>` è **`TreeSet<T>`**.
- L'ordinamento, automatico, è fatto in base alla relazione di ordinamento naturale e viene realizzato **invocando automaticamente il metodo *compareTo***.

Esempio pratico: TreeSet<T>

Esempio d'utilizzo:

```
Set<String> sortedSet = new TreeSet<String>();

sortedSet.add("One");
sortedSet.add("Two");
sortedSet.add("One");
sortedSet.add("Three");

System.out.println(sortedSet.toString());
// (natural order, no duplicates)
// Output: [One, Three, Two]
```

Ciclare su collezioni

Si usa, generalmente, una sintassi diversa per il ***for***. In questo caso si preferisce parlare di *foreach*:

```
List<Libro> libri = new ArrayList<Libro>();

...

for ( Libro c : libri) {
      // qui "c" è una variabile
      // che rappresenta l'i-esimo libro
      System.out.println(c.getAutore());
}
```

"per ogni Libro c nella collezione `libri`**..."**

Ciclare su collezioni

Si usa, generalmente, una sintassi diversa per il *for*, che corrisponde a quella di un *foreach*:

```
Set<String> strings = new HashSet<String>();
...
for ( String s : strings) {
      // qui "s" è una variabile
      // che contiene la stringa i-esima
      System.out.println(s);
}
```

"per ogni Stringa c nella collezione `strings`**..."**

Biblioteca (versione originale)

Si scriva una classe `Biblioteca` che memorizzi le informazioni relative ai libri contenuti all'interno della biblioteca. Occorre memorizzare il nome della biblioteca, l'indirizzo, un codice identificativo univoco (intero). In particolare, occorre memorizzare i libri contenuti nella sezione all'interno di una **lista**.

La classe `Biblioteca` deve inoltre:

1. presentare un opportuno costruttore ~~che prenda come argomento anche il numero massimo di libri che tale biblioteca può contenere~~ (inizialmente la biblioteca non contiene alcun libro).
2. presentare opportuni metodi che permettano di accedere alle variabili di istanza.
3. possedere un metodo `aggiungi` che, dato un oggetto `Libro`, **lo aggiunga all'interno della lista mantenendola ordinata** secondo il punto 5. dell'esercizio precedente (suggerimento: si utilizzi il metodo add(int i, Libro l) della classe List).
4. presentare un metodo `cerca` che, dato il nome di un autore, restituisca un insieme contenente tutti i libri di tale autore all'interno della sezione.
5. possedere un metodo `cancella` che, dato il nome di un autore, rimuova dalla biblioteca tutti i libri di tale autore.
6. possedere il metodo `toString` che restituisca una stringa che fornisca una descrizione della biblioteca, compreso il numero di libri effettivamente presenti.

Biblioteca (versione originale)

```
public class Biblioteca {

        String nome, indirizzo; int codice; List<Libro> libri;

        // 1
        public Biblioteca(String nome, String indirizzo, int codice) {
                this.nome = nome;
                this.indirizzo = indirizzo;
                this.codice = codice;
                libri = new ArrayList<Libro>();
        }
                …
        // 3
        public void aggiungi(Libro l) {
                int i = 0;
                while ((i < libri.size()) && (libri.get(i).compareTo(l) < 0))
                        i++;
                libri.add(i, l);
        }

        // 4
        public Set<Libro> cerca(String autore) {
                Set<Libro> s = new HashSet<Libro>();
                for (Libro l : libri)
                        if (l.getAutore().equals(autore))
                                s.add(l);
                return s;
        }
                …
        // 5
        public void cancella(String autore) {
                for (Libro l : libri)
                        if (l.getAutore().equals(autore))
                                libri.remove(l);
        }
}
```

Applicazione (versione originale)

Si scriva un'applicazione per l'ateneo, che:

1. Crei un vettore di oggetti `Biblioteca` (la cui dimensione deve essere richiesta all'utente).
2. Crei un oggetto `Libro`, lette da tastiera le informazioni necessarie.
3. Trovi la prima biblioteca all'interno del vettore di cui al punto 1. che non contiene alcun libro dell'autore del libro di cui al punto 2.
4. Inserisca il libro di cui al punto 2. nella biblioteca di cui al punto 3 (se tale biblioteca esiste).
5. Letto da tastiera il nome di un autore, provveda a rimuovere i libri di tale autore da tutte le biblioteche.

Applicazione (versione originale)

```
public static void main(String[] args) {

            // 1
            Scanner tastiera = new Scanner(System.in);
            System.out.print("Inserire numero biblioteche: ");
            int num_b = tastiera.nextInt();
            Biblioteca v[] = new Biblioteca[num_b];

            // 2
            System.out.print("Inserire autore: ");
            String autore = tastiera.nextLine();
            System.out.print("Inserire titolo: ");
            String titolo = tastiera.nextLine();
            System.out.print("Inserire casa editrice: ");
            String editore = tastiera.nextLine();
            System.out.print("Inserire anno: ");
            int anno = tastiera.nextInt();
            Libro l = new Libro(autore, titolo, editore, anno);

            // 3
            int i = 0;
            while ((i < v.length) && (!v[i].cerca(l.getAutore()).isEmpty()))
                        i++;

            //4
            if (i < v.length)
                        v[i].aggiungi(l);

            // 5
            autore = tastiera.next();
            for (i = 0; i < v.length; i++)
                        v[i].cancella(autore);
}
```

Esercizio d'esame

Questo esercizio è un esempio di esame effettivamente proposto in un appello passato (16/6/2017)

Esercizio 1 (1/2)

Il comune di Collate (Parma) ha deciso di affidare al dott. Antonio LaSporta la manutenzione degli alberi presenti nelle strade urbane. Per questo, il dott. LaSporta ha deciso di gestire all'interno di un calcolatore le informazioni relative agli alberi presenti sul territorio viabile comunale. In particolare, per ogni albero occorre registrare la matricola, il genere (es. Pino, Abete) e l'anno in cui è stato piantato.

Esercizio 1 (2/2)

Si scriva una classe Albero per il dott. LaSporta che:

1. Possieda un opportuno costruttore con parametri.
2. Presenti opportuni metodi che permettano di accedere alle variabili d'istanza dell'oggetto.
3. Presenti il metodo *toString* che fornisca una descrizione dell'albero.
4. Possieda il metodo *equals* per stabilire l'uguaglianza con un altro oggetto Albero (la verifica va fatta esclusivamente su genere e matricola).
5. Implementi l'interfaccia *Comparable*, definendo il metodo *compareTo* per stabilire la precedenza con un oggetto Albero passato come parametro (per ordine alfabetico del genere e, a parità, per età decrescente).

Esercizio 2 (1/2)

Si scriva una classe Strada che registri le informazioni riguardanti una singola strada del comune di Collate (Parma). Per ogni strada occorre memorizzare il nome e il quartiere, mentre gli alberi vanno inseriti all'interno di un insieme.

La classe Strada deve:

1. Presentare un opportuno costruttore con parametri (inizialmente, l'elenco degli alberi è vuoto).
2. Possedere opportuni metodi che permettano di accedere alle variabili d'istanza dell'oggetto.
3. Presentare il metodo *toString* che fornisca la descrizione della strada (inclusa la descrizione di tutti gli alberi).

Esercizio 2 (2/2)

4. Possedere il metodo *equals* per stabilire l'uguaglianza con un altro oggetto Strada (la verifica va effettuata esclusivamente sul nome).
5. Presentare il metodo *aggiungi* che, dato un oggetto Albero, lo inserisca all'interno dell'insieme, controllando che tale inserimento sia possibile.
6. Possedere il metodo genere che, dato il nome di un genere, restituisca un insieme contenente tutti gli alberi di tale genere inclusi nella strada.
7. Possedere il metodo vecchi che, dato un anno, restituisca il numero totale di alberi contenuti nella strada piantati prima di tale anno.

Esercizio 3 (1/2)

Si scriva un'applicazione per il dott. LaSporta che:

1. Crei una lista di oggetti Strada.
2. Crei un oggetto Strada, lette da tastiera le informazioni necessarie.
3. Inserisca l'oggetto di cui al punto 2. in coda alla lista di cui al punto 1.
4. Crei un oggetto Albero, lette da tastiera le informazioni necessarie.
5. Inserisca l'oggetto di cui al punto 4. all'interno della strada di cui al punto 2., controllando che tale inserimento sia possibile.

Esercizio 3 (2/2)

6. Stampi a video il numero totale di alberi delle strade del comune di Collate (Parma) piantati prima dell'anno 2000.

7. Letto da tastiera il nome di un genere, stampi a video il nome di tutte le strade, all'interno della lista di cui al punto 1., che contengono alberi di tale genere.

8. Stampi a video la descrizione della matricola dell'albero più vecchio, tra quelli del genere di cui al punto 7.