Scorrere una List

- Abbiamo due modi
- Se durante lo scorrimento non dobbiamo eliminare oggetti dalla lista possiamo utilizzare il costrutto **`for-each`**
- Se c'è la necessità di eliminare qualche oggetto mentre si scorre la lista bisogna utilizzare un **iteratore**

Costrutto for-each

```
public String elencoConti(){
   StringBuffer r = new
                        StringBuffer();
   for(BankAccount b : conti)
     r.append(b.toString() + "\n");
   return r.toString();
}
```

Durante il ciclo la variabile **b** punta ogni volta ad un oggetto diverso della lista, nell'ordine degli elementi della stessa

Iteratore

- Un iteratore è un oggetto che permette di scorrere gli elementi di una lista uno alla volta tramite la coppia di metodi
  1) **`hasNext()`** che restituisce **`true`** se c'è ancora un oggetto da scorrere
  2) **`next()`** che restituisce il prossimo oggetto da scorrere
- Si ottiene chiamando il metodo **`iterator()`** su una **`List`**

Iteratore

```
public BankAccount getContoByName(String
  intestatario) {
BankAccount tmp; boolean trovato = false;
Iterator<BankAccount> i = conti.iterator();
while (i.hasNext()){
   tmp = i.next();
   if (tmp.getIntestatario().equals(intestatario))
      return tmp;
}
return null; }
```

Iteratore

- L'iteratore dà la possibilità di rimuovere alcuni oggetti dalla **`List`** durante lo scorrimento
- In seguito ad una chiamata al metodo **`next()`** possiamo effettuare una chiamata al metodo **`remove()`**
- Tale chiamata rimuove dalla **`List`** l'oggetto ottenuto dalla precedente chiamata di **`next()`**
- Non sono ammesse più **`remove()`** in corrispondenza dello stesso **`next()`**

Algoritmi polimorfi su liste

- La classe **`java.util.Collections`** mette a disposizione diversi metodi **statici** polimorfici che operano su liste generiche
- Fra gli altri ci sono **`sort`**, **`shuffle`**, **`reverse`**
- Questi metodi statici prendono una generica **`List<E>`** e modificano l'ordine degli elementi della lista
- **`shuffle`** li mischia a caso
- **`reverse`** inverte la lista

Ordinamento di una List

- Il metodo **`sort`** ordina gli elementi della **`List`** in base ad un certo ordinamento
- Tale metodo opera su liste di oggetti di un generico tipo **`E`**
- Perché tutto funzioni la classe **`E`** deve preoccuparsi di definire l'ordinamento tra gli oggetti
- Esiste un concetto di ordinamento **naturale**

Comparable<E>

- L'interface **`Comparable<E>`** del pacchetto **`java.lang`** richiede a chi vi aderisce l'implementazione di un metodo

```
public int compareTo(E obj)
```

- Le classi involucro e alcune classi delle API (es. **`String`**) implementano questa interfaccia definendo quello che è l'ordinamento naturale fra gli oggetti della classe
- Ad esempio in **`String`** è l'ordinamento lessicografico (ordine alfabetico)

Ordinamento naturale

- Il metodo **`compareTo`** chiamato su certo un oggetto **`obj`**, passando come parametro un altro oggetto **`other`**, deve operare in questo modo:
- Se **`obj`** e **`other`** sono uguali (in accordo con il metodo **`equals`**) deve restituire 0
- Se l'oggetto **`obj`** precede l'oggetto **`other`** nell'ordinamento naturale deve restituire un valore < 0
- Se **`obj`** segue **`other`** deve restituire un valore > 0

Ordinamento naturale

- La definizione del metodo **`compareTo`** per una certa classe deve operare nel modo appena visto e deve garantire che
  - L'ordinamento definito sia totale, cioè il metodo deve funzionare correttamente per qualsiasi coppia di oggetti
  - Sia compatibile con **`equals`**, cioè il metodo deve rispondere 0 per tutte e sole le coppie di oggetti per i quali il metodo **`equals`** risponde **`true`**

Ordinamento di BankAccount

- Ad esempio possiamo decidere che l'ordinamento naturale fra **`BankAccount`** è quello lessicografico fra le stringhe **`idConto`**
- Utilizziamo quindi il metodo **`compareTo`** definito dalla classe **`String`** per implementare il metodo **`compareTo`** fra **`BankAccount`**:

```
public class BankAccount implements
                    Comparable<BankAccount>{
  ...
```

Ordinamento di BankAccount

```
public int compareTo(
                    BankAccount other) {
  // Utilizzo il metodo compareTo fra
  // stringhe
  return idConto.compareTo(
                          other.idConto);
}
```

Ordinamento

```
public String elencoContiPerId() {
  // Ordino la lista in base
  // all'ordinamento naturale
  Collections.sort(conti);
  // Restituisco la lista dei conti
  return elencoConti();
}
```

Comparator<E>

- A volte c'è l'esigenza di ordinare gli elementi di una lista in base ad un ordine diverso rispetto a quello naturale
- Per far questo il pacchetto **`java.util`** definisce una **interfaccia strategica** **`Comparator<E>`**
- A differenza di **`Comparable<E>`**, il metodo richiesto da questa interfaccia è un comparatore diretto fra due oggetti:

```
public int compare(E o1, E o2)
```

Comparatori

- Per usare un ordinamento diverso basta definire una piccola classe che implementa l'interfaccia **`Comparator`** che definisce il metodo **`compare`** in accordo allo stesso
- A questo punto basta creare un oggetto di questa classe e passarlo al metodo polimorfico **`sort(List<E> l, Comparator<E> c)`**
- In genere si definiscono i diversi comparatori come oggetti di una classe interna

Comparatori per BankAccount

- Facciamo tre comparatori:

0) In base al nome dell'intestatario

1) In base alla data di creazione del conto

2) In base al saldo

- Per ognuno di questi definiamo una costante statica della classe **`BankAccount`** a cui assegnamo un oggetto di una classe interna che implementa il relativo **`Comparator`**

Comparatori per BankAccount

```
public static final
  Comparator<BankAccount>
  ORDINA_PER_INTESTATARIO =
   new Comparator<BankAccount>() {
     public int compare(BankAccount b1,
                        BankAccount b2) {
     return b1.intestatario.compareTo(
                        b2.intestatario);
  } };
```

Comparatori per BankAccount

- Si noti che per definire il **`Comparator`** abbiamo utilizzato il metodo **`compareTo`** fra stringhe
- Per definire il **`Comparator`** fra saldo utilizziamo il metodo **`compareTo`** della classe involucro **`Double`**:

Comparatori per BankAccount

```
public static final Comparator<BankAccount>
  ORDINA_PER_SALDO =
    new Comparator<BankAccount>() {
      public int compare(BankAccount b1,
                         BankAccount b2) {
        Double bb1 = new Double(b1.saldo);
        Double bb2 = new Double(b2.saldo);
      return bb1.compareTo(bb2);
    }
};
```

Ordinamento con comparatore

- Possiamo riordinare la lista in base a uno qualunque dei comparatori:

```
public String elencoContiPerSaldo() {
   // Ordino la lista in base al
   //saldo
   Collections.sort(conti,
         BankAccount.ORDINA_PER_SALDO);
   return elencoConti();
}
```

Attenzione

- L'ordinamento definito dai tre comparatori per **`BankAccount`** non è compatibile con **`equals`** di **`BankAccount`**
- Ad esempio due conti diversi intestati alla stessa persona sono diversi secondo **`equals`**, ma il comparatore **`ORDINA_PER_INTESTATARIO`** restituisce 0 in corrispondenza di essi!

Compatibilità con equals

- Utilizzando l'interfaccia **`List`** e i metodi polimorfici su di essa questo non costituisce un grosso problema
- Per altri tipi di **`Collection`**, però, ciò può costituire un grave problema
- È abbastanza semplice rendere i comparatori compatibili con **`equals`**
- Basta fare un confronto in due passi:

Comparator compatibile con equals

```
public static final
  Comparator<BankAccount>
  ORDINA_PER_INTESTATARIO =
   new Comparator<BankAccount>() {
     public int compare(BankAccount b1,
                        BankAccount b2) {
     int cmp = b1.intestatario.compareTo(
                        b2.intestatario);
```

Salvo il valore in una variabile temporanea **cmp**

Comparator compatibile con equals

```
    if (cmp != 0)
      return cmp;
    // Se hanno lo stesso intestatario
    // vale l'ordinamento naturale fra
    // i due conti:
    return b1.compareTo(b2);
}
```

Altre Collection

- L'interface **`List<E>`** fa parte di una **gerarchia di interfacce** per gestire collezioni generiche di oggetti
- In particolare **`List<E>`** è pensata per la gestione di collezioni di oggetti in cui sono possibili ripetizioni e/o si vuole un accesso con indice
- Possiamo mettere lo stesso oggetto più volte in posizioni diverse di una **`List<E>`**
- Possiamo inserire riferimenti **`null`** in una **`List<E>`**

Gerarchia di Collection