Collections

Dagli array alle List
Interfaccia List
Interfaccia Comparable
Interfaccia Comparator
Algoritmi polimorfi sulle liste
Collections e implementazioni
Set, SortedSet

Bank

- Supponiamo di voler definire una classe **`Bank`** che gestisca un certo numero di **`BankAccount`**
- Una **`Bank`** deve permettere di aggiungere un nuovo conto, stampare tutti i conti, ricercare un conto, ecc..
- L'interfaccia pubblica di tale classe è semplice e non presenta problemi
- Un problema rilevante invece si presenta nell'implementazione

Collezione di oggetti

- Per poter fornire i servizi a lei richiesti una **`Bank`** deve poter gestire un insieme di oggetti
- Nello stato di ogni oggetto **`Bank`** deve essere presente una struttura dati che permetta di raggruppare insieme un certo numero di oggetti di una stessa classe
- In questo caso **`BankAccount`**
- L'unica struttura dati che conosciamo finora per far questo è l'array

Stato di Bank

- Dovremo avere nello stato le seguenti variabili istanza

```
private BankAccount[] conti;
// Posizione attuale
private int posizioneAttuale;
...
```

Costruttore

- All'atto della creazione di una nuova **`Bank`** dovremo decidere quanti conti questa potrà gestire
- Questo perché un array può essere creato di qualsiasi dimensione, ma questa non è espandibile successivamente
- Ciò impone delle restrizioni sul comportamento della **`Bank`**
- Inoltre siamo costretti a gestire il controllo del numero di conti

Costruttore di Bank

```
public Bank(String nome, String
  internationalCode, int dimensione) {
     ...
     conti = new
             BankAccount[dimensione];
     posizioneAttuale = 0;
  }
```

Controlli su Bank

```
public BankAccount nuovoConto(String
         intestatario, double initialBalance)
throws DimensioneBancaInsufficienteException
  {
    if (posizioneAttuale == conti.length)
      throw new
   DimensioneBancaInsufficienteException("Si
  vogliono inserire più di " + conti.length +
  " conti, che è la dimensione massima.");
    else { ... // Crea il conto
```

Per la ricerca di un conto

- Dobbiamo istanziare lo schema della ricerca lineare incerta
- Com'è fatta la condizione di ricerca?
- Può essere basata, ad esempio, sui valori di alcune variabili istanza dell'oggetto
- Quando si gestisce una collezione di oggetti è sempre bene specificare una relazione di uguaglianza fra gli oggetti ridefinendo il metodo **`equals`**

Ridefinizione di equals

- L'uguaglianza fra oggetti di una nuova classe va definita sempre utilizzando l'uguaglianza fra i valori di **alcune** variabili istanza
- È bene scegliere le variabili istanza i cui valori rimangono **immutati** durante la vita normale dell'oggetto
- Questo perché l'uguaglianza non deve cambiare durante la vita di un oggetto
- Queste variabili istanza dovrebbero essere dichiarate **`final`**

Uguaglianza fra conti

- Aggiungiamo allo stato di **`BankAccount`** una stringa che è l'identificatore del conto corrente
- Poniamola **`final`**
- Questa variabile potrà essere assegnata soltanto in un costruttore
- Come criterio di uguaglianza fra **`BankAccount`** utilizziamo l'uguaglianza fra questi identificatori (uguaglianza tra stringhe)

Stato di BankAccount

```
public class BankAccount {
  // Si suppone sia un identificativo, il
  // metodo equals usa questo valore
  // per confrontare gli oggetti
  private final String idConto;
  // Intestatario del conto
  private String intestatario;
  private GregorianCalendar dataCreazione;
  private double saldo;
```

Ridefinizione di equals

```
public boolean equals(Object o) {
   if (!(o instanceof BankAccount))
            return false;
   BankAccount other = (BankAccount) o;
   return idConto.equals(other.idConto);
}
```

Codice hash di BankAccount

- L'uguaglianza si basa solo sulla variabile istanza **`idConto`**, che è una stringa
- Quindi in questo caso semplice non occorre calcolare l'**`hashCode`** con l'algoritmo generale
- Utilizziamo l'hash fornito dalla classe **`String`**

```
public int hashCode(){
        return idConto.hashCode();
    }
```

Ricerca di un BankAccount

- Avendo ridefinito in questo modo il metodo **`equals`** possiamo definire la condizione della ricerca lineare incerta come segue:
- Creiamo un oggetto **`BankAccount`** fittizio in cui l'unico valore significativo è quello della variabile istanza **`idConto`**
- La condizione di ricerca sarà che questo conto fittizio deve risultare uguale al conto in posizione **`i`** dell'array, secondo il metodo **`equals`**

Ricerca di un BankAccount

```
public int search(String idConto) {
  // Conto fittizio per il confronto
  BankAccount confronto = new
                  BankAccount(idConto, "", 0);
  // Ricerca lineare incerta
  boolean trovato = false; int i = 0;
  while (i < posizioneAttuale && !trovato)
    if (conti[i].equals(confronto))
      trovato = true;
    else i++;
```

Rimozione di un BankAccount

```
public boolean remove(String idConto) {
    int pos = search(idConto);
    if (pos == -1)
      return false;
    for (int i = pos; i < posizioneAttuale -1; i++)
      conti[i] = conti[i+1];
    conti[posizioneAttuale - 1] = null;
    posizioneAttuale--;
    return true;
    }
```

Prima ricerca
la posizione

Poi ricompatta l'array
spostando di una posizione
verso sinistra tutti i conti
successivi a quello cancellato

Dall'array alla List

- Nel pacchetto **`java.util`** è definita l'**interface `List<E>`**
- Oggetti di classi che implementano questa interfaccia possono collezionare in una lista (sequenza) un insieme di oggetti di una stessa classe
- **`<E>`** rappresenta un generico tipo (nome di classe)
- L'interface **`List<E>`** mette a disposizione metodi molto interessanti

Interface List<E>

```
public interface List<E> extends Collection<E> {
    // Accesso
    E get(int index);
    E set(int index, E element);     // Optional
    boolean add(E element);          // Optional
    void add(int index, E element); // Optional
    E remove(int index);             // Optional
```

Optional si riferisce al fatto che alcune implementazioni potrebbero non fornire quel servizio, lanciando un'eccezione

Uso di List<E>

- Un oggetto con queste caratteristiche può essere visto come un array "evoluto"
- Supponiamo di avere un oggetto **`myList`** di questo tipo
- Il metodo **`myList.get(i)`** può essere visto come l'equivalente dell'accesso ad un array con la notazione **`myList[i]`**
- Allo stesso modo **`myList.set(i, o);`** è l'analogo di **`myList[i] = o;`**

Uso di List<E>

- Una caratteristica importante si deduce dal metodo **`add(o)`**
- Tale metodo aggiunge l'oggetto **`o`** alla fine della lista
- Non c'è un limite alla dimensione
- La capacità della lista si adatta automaticamente alla necessità (finché c'è memoria disponibile)
- Il metodo **`add(i, o)`** mette l'oggetto **`o`** in posizione **`i`** spostando verso destra tutti gli oggetti in posizione >= **`i`**

Bank

- Invece dell'array di **`BankAccount`** possiamo utilizzare una **`List`**

```
public class Bank {
  private String nome;
  // Codice internazionale
  private String internationalCode;
  // Utilizzo una List per mantenere i conti
  private List<BankAccount> conti;
  ...
```

Creazione di una List

- **`List<E>`** è una **`interface`** e quindi non possiamo creare direttamente oggetti di questo tipo
- Il pacchetto **`java.util`** mette a disposizione alcune classi che implementano questa interfaccia
- La classe standard è **`ArrayList<E>`**
- Se si usa una **`List`** è bene utilizzare variabili polimorfe e assegnarle con un oggetto di una certa classe solo al momento della creazione
- Ciò rende il nostro codice più generico e indipendente dall'implementazione della **`List`**

Costruttore di Bank

```
public Bank(String nome, String
                    internationalCode) {
  this.nome = nome;
  this.internationalCode =
                     internationalCode;
  // Utilizzo l'implementazione ArrayList
  conti = new ArrayList<BankAccount>();
}
```

Nuovo conto

```
public BankAccount nuovoConto(String
  intestatario, double initialBalance) {
  // Costruisco l'ID
  String idConto = internationalCode + id;
  id++;
  // Creo il nuovo conto e lo aggiungo alla lista
  // dei conti
  BankAccount tmp = new BankAccount(idConto,
             intestatario, initialBalance);
  conti.add(tmp);
  return tmp;
```

Nessun controllo richiesto

Ricerca

- L'interface `List<E>` mette a disposizione anche un algoritmo polimorfo di ricerca
- Esso si basa sul metodo `equals` degli oggetti che fanno parte della `List`
- Basta ridefinirlo in modo opportuno e non dobbiamo più scrivere noi il codice per la ricerca lineare incerta

Ricerca

```
public int search(String idConto) {
  // Conto fittizio per il
  confronto // tramite il metodo
  equals
  BankAccount confronto = new
        BankAccount(idConto, "", 0);
 // Ricerca tramite il metodo di List
  return conti.indexOf(confronto);
}
```

Ricerca lineare incerta basata su **`equals`** di **`BankAccount`**

Rimozione di un conto

```
public boolean remove(String idConto) {
    // Cerca il conto
    int pos = search(idConto);
    if (pos == -1)
      return false;
    // Elimita l'elemento dalla lista dei conti
    conti.remove(pos);
    return true;
    }
```

Gerarchia Collections

- Il diagramma UML delle interfacce, classi astratte e classi della gerarchia delle collections si può trovare (ad esempio) su:
- https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram

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**

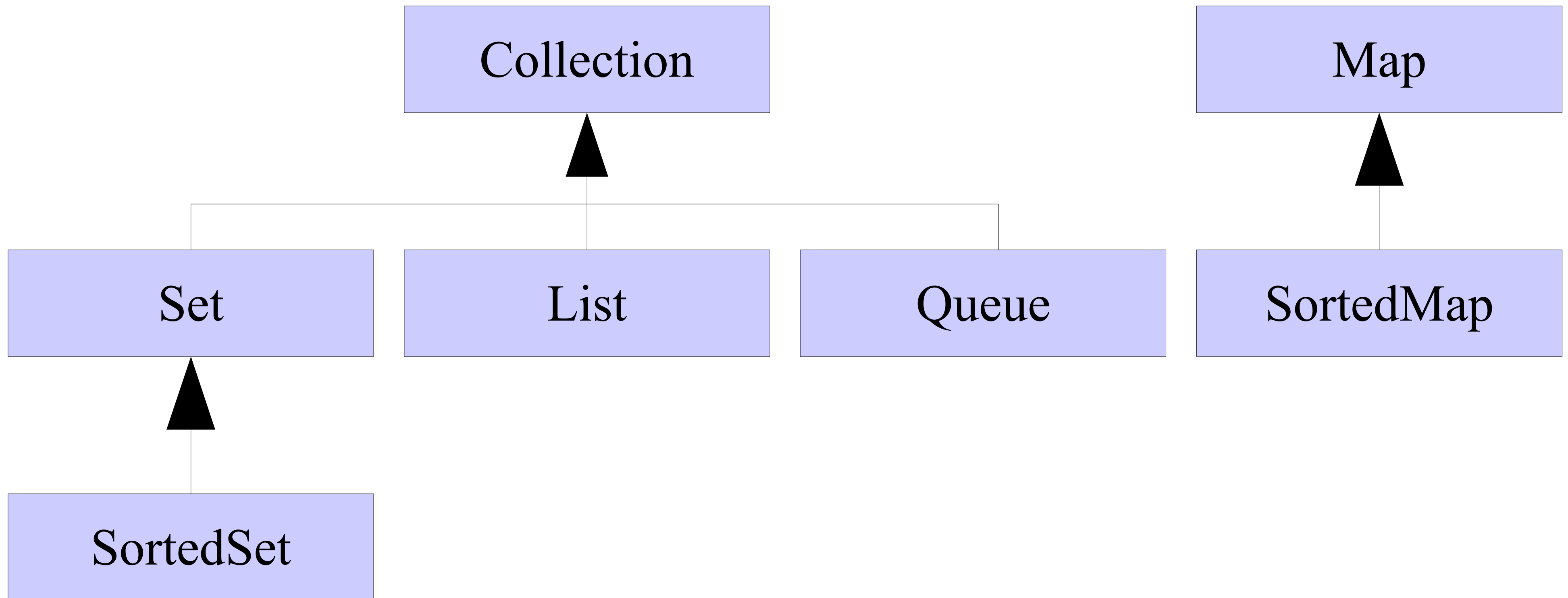
Comparator compatibile con equals

```
  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



Gerarchia di Collection

- Vediamo un po' più in dettaglio solo **`Set<E>`** e **`SortedSet<E>`**
- Per le altre interfacce si possono consultare le API o il tutorial sulle collections (sul sito **`http://java.sun.com`**)

Set<E>

- Abbiamo visto **`List<E>`** per mantenere una collezione di oggetti con ripetizione
- L'interface **`Set<E>`** invece modella esattamente il **concetto matematico di insieme**
- Rappresenta una collezione di oggetti di tipo **`E`** in cui un oggetto è presente una sola volta oppure non è presente
- Se si aggiunge un oggetto già presente il **`Set`** non cambia

Interfaccia di Set<E>

```
public interface Set<E> extends Collection<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();
    ...
```

Appartenenza

- Il metodo **`contains`** è l'analogo di **`indexOf`** di **`List<E>`**
- Qui restituisce **`true`** se presente e **`false`** se non presente, dato che **in un insieme non c'è il concetto di posizione**
- Valgono le stesse considerazioni fatte riguardo alla ridefinizione del metodo **`equals`**: l'appartenenza viene stabilita scorrendo gli elementi dell'insieme e confrontandoli, tramite il metodo **`equals`**, con l'oggetto passato

Inserimento e rimozione

- I metodi **`add`** e **`remove`** restituiscono un **`boolean`** che indica se l'operazione ha modificato il **`Set<E>`**
- Nel caso di **`add`** l'aggiunta di un oggetto già presente (secondo il metodo **`equals`** della classe **`E`**) non cambia il **`Set<E>`**
- Nel caso di **`remove`** l'eliminazione di un oggetto non presente (secondo il metodo **`equals`** della classe) non cambia il **`Set<E>`**
- L'iteratore è analogo a quello di **`List<E>`** tranne che qui l'ordine non ha un significato particolare

Classe standard di implementazione

- La classe standard del pacchetto **`java.util`** che implementa l'interface **`Set<E>`** è la classe **`HashSet<E>`**
- Fa uso di tecniche di hash: è quindi assolutamente necessario ridefinire correttamente il metodo **`hashCode`** della classe **`E`** se si ridefinisce **`equals`**
- Una classe alternativa è **`TreeSet<E>`** che usa alberi particolari che mantengono un ordinamento

SortedSet<E>

- L'interfaccia **`SortedSet<E>`** modella un insieme in cui **gli elementi hanno un ordinamento**
- Vale qui il concetto di ordinamento naturale espresso dall'interfaccia **`Comparable<E>`** che abbiamo visto anche in **`List<E>`**
- Però se si vuole utilizzare **`SortedSet<E>`** è assolutamente necessario che l'ordinamento naturale di **`E`** sia compatibile con **`equals`** di **`E`**

Interface SortedSet<E>

```
public interface SortedSet<E> extends Set<E> {
    //Tutti i metodi di Set<E> più:
    SortedSet<E> subSet(E fromElement,
                                E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    E first(); // Minimo
    E last();  // Massimo
```

Classe implementazione

- La classe di implementazione standard per `SortedSet<E>` è `TreeSet<E>`
- L'iteratore di un `SortedSet<E>` scorre gli elementi dell'insieme **secondo l'ordinamento naturale** di `E`, dal più piccolo al più grande
- Un punto di forza dell'uso di questa interfaccia sta nel fatto che durante una ricerca possiamo **evitare di iterare su tutti gli elementi** fermandoci quando abbiamo oltrepassato il punto dove dovrebbe essere l'elemento cercato secondo l'ordinamento naturale

Esempio: prenotazioni di aule

- Modelliamo un dominio del discorso in cui una comunità di utenti ha a disposizione un certo insieme di aule con certe caratteristiche
- Ogni utente può prenotare un'aula per un certo giorno, in una certa ora, per una certa attività
- Non ci devono essere sovrapposizioni nelle prenotazioni
- Definiamo una classe **`GestoreAule`** che mantiene un **`Set<Aula>`** contenente le aule

GestoreAule

```
public class GestoreAule {
  private Set<Aula> aule;
  public GestoreAule() {
    aule = new HashSet<Aula>();
  }
  public boolean add(Aula a){
    if (a == null)
      return false;
    return aule.add(a);
  } ...
```

Usiamo l'implementazione standard

L'implementazione standard permette di inserire in un **`Set`** anche elementi **`null`**
In questo caso facciamo un controllo per evitarlo

Prenotazione

- Una prenotazione viene fatta da un certo utente, in una certa data e ora per una certa attività

```
public class Prenotazione implements
  Comparable<Prenotazione> {

  // L'utente che effettua questa prenotazione

  private Utente utente;

  // Data e ora della prenotazione

  private DataEOra dataEOra;

  private String descrizione; ...
```

Prenotazione

- La classe **`Prenotazione`** implementa l'interfaccia **`Comparable<Prenotazione>`**
- L'ordinamento naturale che definisce è quello cronologico
- Una prenotazione precede un'altra se è in una data e ora precedente
- Questo ci permette di usare un **`SortedSet<Prenotazioni>`** per mantenere le prenotazioni di **`Aula`** ordinate per data e ora

Aula

- Ogni Aula ha il suo personale insieme di prenotazioni ordinato cronologicamente:

```
public class Aula {
  // Identificativo unico di un'aula
  private final String nome;
  ...
  // Insieme delle prenotazioni per
  quest'aula, // in ordine cronologico
  private SortedSet<Prenotazione> prenotazioni;
  ...
```

Controllo Aula libera

- Per controllare se un'**`Aula`** è libera in una certa data e ora dovremmo esaminare tutte le prenotazioni dell'**`Aula`** e verificare che non ce n'è nessuna in quella data e in quella ora
- Usando l'ordinamento cronologico dell'insieme possiamo organizzare la ricerca in modo tale da evitare di guardare tutte le prenotazioni se non è necessario
- Basterà fermarsi dopo il punto in cui dovrebbe trovarsi la prenotazione, se ci fosse

Controllo Aula libera

```
public boolean libera(DataEOra d) {
  Prenotazione p;
  DataEOra tmp;
  // Scorre l'insieme ordinato di prenotazioni
 e // vede se la data e l'ora sono occupate
  Iterator<Prenotazione> i =
                   prenotazioni.iterator();
  // Cerco una prenotazione nella stessa data
 e // nella stessa ora
  // Continua...
```

Controllo Aula libera

```
boolean trovato = false;
boolean interrompi = false;
while (i.hasNext() && !trovato && !interrompi){
  p = i.next(); tmp = p.getDataEOra();
  if (tmp.equals(d))
     trovato = true;
  else if (tmp.compareTo(d) > 0)
       // Ho superato la data senza aver trovato
       interrompi = true;
   } // Continua...
```

Controllo Aula libera

```
if (trovato)
        return false;
else return true;
}
```