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

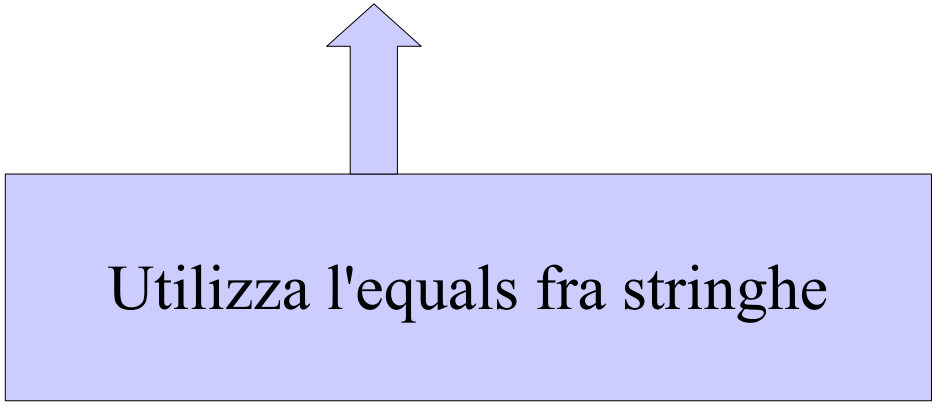
- Aggiungiamo allo stato di **`BankAccount`** una stringa che è l'identificatore del conto corrente
- Poniamola **`final`**
- Questa variabile potrà essere assegnata soltanto in un costruttore
- Come criterio di uguaglianza fra **`BankAccount`** utilizziamo l'uguaglianza fra questi identificatori (uguaglianza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