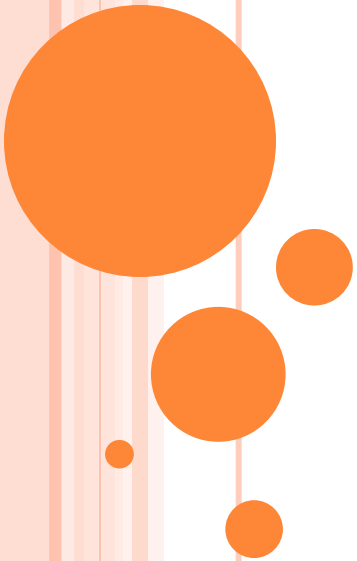
COLLECTION FRAMEWORK



LE INTERFACCE

INTERFACCE

- Le interfacce sono uno strumento molto importante per il design e l'implementazione di sistemi **OO**.
- In queste slide approfondiremo le interfacce in Java e il loro utilizzo per migliorare la riusabilità e l'estendibilità del software

INTERFACCE

- Un **metodo astratto è un metodo senza** corpo, con un "**;**" dopo l'intestazione
- Una **interfaccia (interface) in Java ha una** struttura simile a una classe, ma può contenere SOLO **costanti e metodi d'istanza astratti** (quindi non può contenere né *costruttori, né variabili statiche, né variabili di istanza, né metodi statici).*

INTRODUZIONE

- Una delle linee guida fondamentali nella progettazione OO è la separazione fra l'interfaccia di una classe e la sua implementazione.
- A tal proposito i progettisti di Java hanno dotato il linguaggio di un costrutto, l'interfaccia appunto, distinto dalla classe.
- In questo modo si hanno, a livello di implementazione, due strumenti distinti per definire gli aspetti comportamentali (interfaccia) e quelli implementativi (classe).

JAVA.LANG.COMPARABLE

```
public interface Comparable {
      public int compareTo(Object o);
}
```

- Spesso l'interfaccia comprende anche una descrizione **informale** del significato dei metodi (che chiameremo **specifica o contratto).**

COMPARABLE

- Ad esempio, nella API di **Comparable** si vede che al metodo **compareTo** è associata una precisa interpretazione (*descritta a parole*): esso definisce un ordinamento totale sugli oggetti della classe.
- Estratto:
  - **compareTo:** Compares **this** object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. [...] The implementor must also ensure that the relation is transitive: **(x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.** [...] It is strongly recommended, but not strictly required that **(x.compareTo(y)==0) == (x.equals(y)). [...]**

IMPLEMENTARE UN'INTERFACCIA

- Si può dichiarare che una classe **implementa (implements)** una data interfaccia: in questo caso la classe deve fornire una **realizzazione per tutti i metodi** astratti dell'interfaccia, cioè la classe deve fornire metodi con la stessa firma descritta nell'interfaccia (**e con il corpo, naturalmente**).
- La realizzazione di un metodo deve anche rispettare la "specifica" del corrispondente metodo astratto.

ESEMPIO: INTERI

```
public class Intero implements Comparable {
[…]
      public int compareTo(Object o) {
            Intero int1 = (Intero) o;
            if (n < int1.n) return -1;
            else if (n > int1.n) return 1;
            else return 0;
      }[…]
}
```

A COSA SERVONO LE INTERFACCE

- per definire **Tipi di Dati Astratti** si pensi diverse possibili implementazioni di ADT (*liste con/senza ripetizioni, array, tabelle hash, alberi binari di ricerca,…*);
- come **contratto** tra chi implementa una classe e chi la usa: le due parti possono essere sviluppate e compilate separatamente;
  - utile per i gruppi di lavoro
- per scrivere **programmi generici** (applicabili a più classi), evitando di duplicare il codice;
  - Ad esempio, sfruttando l'interfaccia **Comparable**, si possono scrivere algoritmi di ordinamento generici, applicabili a istanze di qualunque classe che la implementi.

REGOLE DI UTILIZZO

- Possiamo dichiarare una variabile indicando come tipo un'interfaccia: **Comparable cmp;**
- Non possiamo istanziare un'interfaccia:

  Comparable com = new Comparable();

  // VIETATO
- Ad una variabile di tipo interfaccia possiamo assegnare solo istanze di classi che implementano l'interfaccia:

  Comparable com = new Intero(5);
- Su di una variabile di tipo interfaccia possiamo invocare solo metodi dichiarati nell'interfaccia (o nelle sue "super-interfacce").

L'INTERFACCIA RUNNABLE

- Vediamo un semplice esempio dalla libreria di Java, l'interfaccia Runnable:

```
public interface Runnable {
    public abstract void run();
}
```

- *The Runnable interface should be implemented by **any class whose instances are intended to be executed by a thread**. The class must define a method of no arguments called **run**.*
- *This interface is designed to provide a common protocol for objects that **wish to execute code while they are active**. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.*
  http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Runnable.html

L'INTERFACCIA RUNNABLE

- Una classe che voglia implementare l'interfaccia **Runnable** dovrà definire un metodo **run()** che conterrà il codice da eseguire in un thread di esecuzione separato.
- Una classe client che debba mandare in esecuzione un thread conterrà codice simile al seguente

L'INTERFACCIA RUNNABLE

```
public class Test {
    public void runIt(Runnable obj) {
        ....
        obj.run();
        ....
    }
}
```

VANTAGGI E SVANTAGGI

- Il principale vantaggio derivante dall'utilizzo dell'interfaccia **Runnable** nel codice precedente consiste nel fatto che **il metodo runIt() accetta come parametro oggetti di classe diversa**, senza alcun legame fra di loro se non il fatto che tutti devono implementare l'interfaccia Runnable.

VANTAGGI E SVANTAGGI

- Il metodo runIt() è quindi ampiamente riutilizzabile e inoltre viene garantita anche la massima estendibilità
- non si ha, infatti, alcun vincolo sulle classi che implementano l'interfaccia e nulla vieta, ad esempio, di includere funzionalità avanzate come il *pooling di thread*.

VANTAGGI E SVANTAGGI

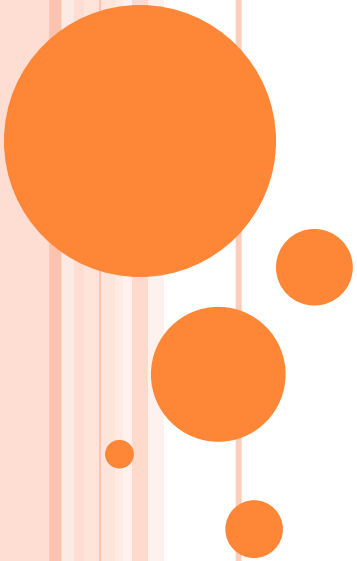
- Il legame fra una classe che implementa un'interfaccia e i suoi client è rappresentato da i parametri dei suoi metodi.
- Per avere il massimo disaccoppiamento occorre quindi fare in modo che i parametri delle interfacce siano tipi predefiniti oppure interfacce, ma non classi concrete.

VANTAGGI E SVANTAGGI

- L'introduzione delle interfacce nel design permette, quindi, di ridurre le dipendenze da classi concrete ed è alla base di uno dei principi fondamentali della programmazione a oggetti: "*Program to an interface, not an implementation*“
- E' importante a questo punto osservare **come un'interfaccia rappresenti un contratto fra la classe che la implementa e i suoi client.**

VANTAGGI E SVANTAGGI

- I termini del contratto sono i metodi dichiarati nell'interfaccia, metodi che ogni classe che implementi l'interfaccia si impegna a definire.
- Sul comportamento dei metodi però non è possibile specificare nulla in Java e altri linguaggi come C++ se non attraverso la documentazione.



UTILIZZO DELLE INTERFACCE

Approfondimento

UTILIZZI DELLE INTERFACCE

- Vediamo ora alcuni utilizzi tipici delle interfacce in Java e ne analizzeremo i vantaggi a livello di design e di implementazione.
- Inoltre considereremo gli aspetti di ereditarietà fra interfacce e di creazione di oggetti dei quali è nota solo l'interfaccia.

INTERFACCE E POLIMORFISMO

- Il polimorfismo ottenuto attraverso l'ereditarietà è uno strumento molto potente.
- Le interfacce ci permettono di sfruttare il polimorfismo anche senza ricorrere a gerarchie di ereditarietà.

```
public interface PricedItem {
    public void setPrice(double price);
    public double getPrice();
}
```

- L'interfaccia PricedItem definisce il comportamento di un articolo con prezzo.
- Possiamo a questo punto implementare l'interfaccia in una classe applicativa Book nel modo seguente

```
public class Book implements PricedItem {
      private String title;
      private String author;
      private double price;
      ...
      public void setPrice(double price) {
          this.price = price;
      }
      public double getPrice() {
          return price;
      }
  }
```

- Il codice client che necessita del prezzo di un libro può quindi essere scritto così:

```
double computeTotalPrice(Collection items) {
    Iterator it = items.iterator();
    PricedItem item;
    double total = 0;
    while (it.hasNext()) {
        item = (PricedItem)it.next();
        total += item.getPrice();
    }
    return total;
}
```

- Il metodo precedente calcola il prezzo totale di una collezione di oggetti. Supponiamo ora di voler estendere l'applicazione per gestire non solo libri ma anche CD musicali. Introdurremo a questo proposito una nuova classe che implementa l'interfaccia PricedItem

```
public class CD implements PricedItem {
    private String title;
    private String singer;
    private Collection songs;
    private double price;
    ....
    public void setPrice(double price) {
        this.price = price;
    }
    public double getPrice() {
        return price;
    }
}
```

- Il codice precedente per il calcolo del prezzo totale funziona senza modifiche anche se la collezione contiene oggetti di classe CD perchè tale codice fà riferimento all'interfaccia e non alla classe concreta.

"EREDITARIETÀ" MULTIPLA

- Come noto Java non supporta l'ereditarietà multipla fra classi. Una classe può però implementare più interfacce e in questo modo possiamo per essa definire diversi comportamenti. Riprendiamo ora l'esempio precedente e aggiungiamo alle nostre classi il supporto alla persistenza. L'interfaccia Persistent fà al caso nostro

```
public interface Persistent {
        public void save();
    }
```

- Le nostre classi diventano quindi

```
    public class Book implements PricedItem, Persistable
{
        ...
    }
    public class CD implements PricedItem, Persistable {
        ...
    }
```

- Quindi possiamo scrivere il codice di gestione del salvataggio nel seguente modo

```
public void saveAll(Collection items) {
     Iterator it = items.iterator();
     Persistent item;
     while (it.hasNext()) {
          item = (Persistent)it.next();
          item.save();
     }
}
```

- Osserviamo che l'interfaccia Persistent nasconde completamente i dettagli di salvataggio che potrebbe avvenire su file oppure su DB attraverso JDBC.

COMPOSIZIONE

- La programmazione OO permette di riutilizzare funzionalità esistenti principalmente attraverso ereditarietà fra classi e composizione di oggetti.
- La composizione permette di ottenere sistemi più flessibili mentre l'ereditarietà dovrebbe essere utilizzata principalmente per modellare relazioni costanti nel tempo

- Non dobbiamo però pensare di poter ottenere il polimorfismo solo con l'ereditarietà:
  - l'utilizzo combinato di interfacce e composizione ci permette di progettare soluzioni molto interessanti dal punto di vista architetturale.

- Vediamo un esempio.
- Supponiamo di dover sviluppare il supporto alla validazione per le classi Book viste prima.
- Le logiche di validazione saranno incorporate all'interno di un'opportuna classe che implementa una interfaccia Validator

```
public interface Validator {
      public void validate(Object o);
  }
public class BookValidator implements Validator {

      public void validate(Object o) {
          if (o instanceof Book) {
              ...
          }
      }
  }
```

- Vediamo ora la classe che si occupa di eseguire la validazione

```
public class Manager {
    ...
    Validator validator;
    public Manager(Validator validator) {
        this.validator = validator;
        ...
    }
    public void validate() {
        ...
        validator.validate(object)
        ...
    }
}
```

- La classe Manager non fà riferimento alla classe concreta BookValidator quindi possiamo cambiare la logica di validazione anche a run-time.
- La soluzione di design che abbiamo visto è nota come pattern Stategy

INTERFACCE CHE ESTENDONO ALTRE INTERFACCE

- Come le classi anche le interfacce possono essere organizzate in gerarchie di ereditarietà.
- Ad esempio

```
interface Base {
    ...
}
  interface Extended extends Base {
    ...
}
```

- L'interfaccia Extended eredita quindi tutte le costanti e tutti i metodi dichiarati in Base. Ogni classe che implementa Extended dovrà quindi fornire una definizione anche per i metodi dichiarati in Base. Le interfacce possono poi derivare da più interfacce, allo stesso modo in cui una classe può implementare più interfacce.

```
interface Sibling { ...}
    interface Multiple extends Base, Sibling
{ ... }
```

- Vediamo ora come vengono gestiti i conflitti di nomi.
- Se due interfacce contengono due metodi con la stessa signature e con lo stesso valore di ritorno allora la classe concreta dovrà implementare il metodo solo una volta e il compilatore non segnalerà alcun errore.

- Se i metodi hanno invece lo stesso nome ma signature diverse allora la classe concreta dovrà dare un'implementazione per entrambi i metodi.
- I problemi si verificano quando le interfacce dichiarano due metodi con gli stessi parametri ma diverso valore di ritorno. Es.

```
interface Int1 {
      int foo();
  }
  interface Int2 {
      String foo();
  }
```

- In questo caso il compilatore segnala un errore perchè il linguaggio non permette che una classe abbia due metodi la cui signature differisce solo per il tipo del valore di ritorno.
- Consideriamo infine il caso in cui due interfacce dichiarino due costanti con lo stesso nome, eventualmente anche con tipo diverso.
- La classe concreta potrà utilizzare entrambe le costanti qualificandole con il nome dell'interfaccia.

INTERFACCE E CREAZIONE DI OGGETTI

- Come abbiamo visto le interfacce permettono di astrarre dai dettagli implementativi, eliminare le dipendenze da classi concrete e porre l'attenzione sul ruolo degli oggetti nell'architettura che si vuole sviluppare.
- Rimane però un problema relativo alla creazione degli oggetti: in tale situazione occorre comunque specificare il nome di una classe concreta.

- In questo caso si genera quindi una dipendenza di creazione
- Ad esempio

```
public class MyDocument {
        ....
        public void open();
        public void close();
        public void save();
        ....
    }
    MyDocument doc = new MyDocument();
```

- Nell'istruzione precedente si crea un oggetto di classe concreta MyDocument ma il codice non potrà essere riutilizzato per creare un oggetto di classe estesa da MyDocument oppure un'altra classe che rappresenta un diverso tipo di documento.
- Come osservato è possibile risolvere questo problema creando classi final oppure rendendo ridefinibile l'operazione di creazione.

- Quest'ultima soluzione è senz'altro la più interessante e i pattern creazionali ci permettono di risolvere il problema.
- Vediamo ora come applicare il pattern Abstract Factory per incapsulare il processo di creazione ed eliminare la dipendenze di creazione di cui soffriva il precedente esempio.
- Innanzi tutto introduciamo un'interfaccia per rappresentare un documento

```
public interface Document {
      public void open();
      public void close();
      public void save();
  }
```

- A questo punto definiamo un'interfaccia per un factory, cioè un oggetto il cui compito è quello di creare altri oggetti.
- Poichè a priori non sappiamo quale tipo di oggetti dovranno essere creati ricorriamo alle interfacce

```
public interface DocumentFacory {
        public Document createDocument();
    }
```

- Possiamo ora definire diversi tipi di documento, ad esempio

```
public class TechicalDocument implements Document {
     public void open() { ... }
     public void close() { ... }
     public void save() { ... }
 }
 public class CommercialDocument implements Document {
     public void open() { ... }
     public void close() { ... }
     public void save() { ... }
 }
```

- Ora vogliamo essere in grado di creare diversi tipi di documento.
- Per questo definiamo una classe factory per ogni diversa classe documento

```
    public class TechicalDocumentFactory implements
DocumentFactory {
        public Document createDocument() {
            Document doc = new TechicalDocument();
            ...
            return doc;
        }
    }
    public class CommercialDocumentFactory implements
DocumentFactory {
        public Document createDocument() {
            Document doc = new CommercialDocument();
            ...
            return doc;
        }
    }
```

- Possiamo quindi creare oggetti documento nel modo seguente

```
void manageDocument(DocumentFactory factory) {
    Document doc = factory.createDocument();
    doc.open();
    ...// modify document
    doc.save();
    doc.close();
}
```

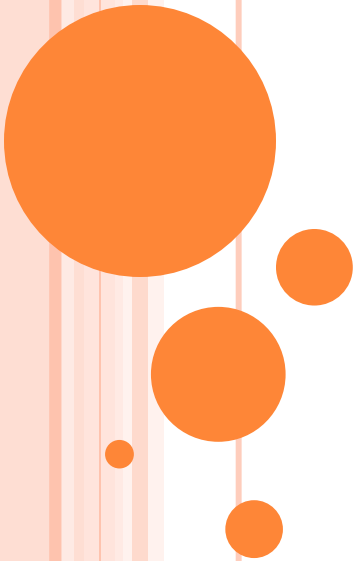
- Il codice precedente crea un oggetto che implementa l'interfaccia Document ma non ha alcun legame con classi concrete e si può quindi utilizzare con classi diverse, purché conformi all'interfaccia Document.

VANTAGGI DELLE INTERFACCE NELLO SVILUPPO DEL SOFTWARE

- Dopo aver passato in rassegna diversi esempi sull'utilizzo delle interfacce possiamo a questo punto discutere sui loro reali vantaggi:
  - le interfacce permettono innanzitutto di concentrarsi sugli aspetti comportamentali degli oggetti e costruire quindi astrazioni efficaci per il problema in esame, nascondendo i dettagli implementativi all'interno delle classi concrete;
  - ragionare per interfacce permette di separare le politiche di interazione fra classi dalle caratteristiche interne di una classe;

VANTAGGI DELLE INTERFACCE NELLO SVILUPPO DEL SOFTWARE

- rappresentano inoltre uno strumento per il disaccoppiamento fra classi concrete, ovvero per l'eleminazione delle dipendenze che abbiamo visto essere deleterie per un buon design.



INTERFACCIA COLLECTION

L'INTERFACCIA COLLECTION

- Dalle API:
  - **Interface Collection**: the root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered.
- Idea: mettere in una interfaccia tutte le caratteristiche comuni degli insiemi di oggetti (ordinati o no, con duplicazioni o no).

METODI DI COLLECTION (1)

- I più importanti:
  - **add(Object o)** aggiungi un oggetto a una collezione
  - **boolean contains(Object o)** vede se un oggetto è contenuto nella collezione
  - **boolean isEmpty()** verifica se la collezione è vuota
  - **Iterator iterator()** fornisce un iteratore per fare una scansione della collezione

METODI DI COLLECTION (2)

- **boolean remove(Object o)** rimuove un elemento dalla collezione
- **int size()** numero di elementi della collezione

NOTA SUL METODO `REMOVE`

- Il metodo remove ritorna un booleano
- Questo booleano dovrebbe valere **True** se la collezione cambia come risultato della remove (l'elemento da eliminare era presente ed è stato eliminato)

CONTRATTO E FIRMA

- Dei metodi delle interfacce si distinguono:
  - **firma** : l'intestazione del metodo
  - **contratto** : una definizione (a parole, in inglese *preferibilmente*) di cosa si intende che faccia il metodo

- Per esempio, il contratto di *remove* è che si tratta di un metodo che rimuove un elemento e che il valore di ritorno indica se la rimozione è stata effettuata o meno

CONTRATTO E FIRMA

- La firma dice soltanto che è un metodo che ha un Object come argomento e un **boolean** come valore di ritorno
- In Java si può solo specificare la firma
- Il contratto viene dato a parole e non è obbligatorio
  - è consigliato, in particolar modo, nei lavori di gruppo

CONTRATTO E FIRMA

- Se non si rispetta il contratto o la firma:

Non si rispetta	quando per esempio	cosa succede
firma	viene implementato un metodo `remove` che non ha come argomento un `Object` o un valore di ritorno `boolean`	la classe non implementa l'interfaccia, e quindi viene segnalato un errore
contratto	il metodo `remove` ha la firma giusta, ma invece di rimuovere l'elemento lo aggiunge	chi usa la classe non riesce a far funzionare i suoi programmi

- Non rispettare la firma è un errore di linguaggio
- Non rispettare il contratto è un errore semantico
  - entrambi sono molto gravi

COSTRUTTORI

- Le interfacce non possono contenere costruttori (è nel linguaggio)
- Il contratto dell'interfaccia **Collection** specifica però che ci deve essere un costruttore vuoto, che crea la collezione senza elementi

ESEMPIO DI IMPLEMENTAZIONE

- Le classi che implementano direttamente *Collection* rappresentano **multi-insiemi non ordinati di oggetti.**
- Esempio di implementazione: usiamo una lista.

ESEMPIO DI IMPLEMENTAZIONE

```
import java.util.*;

class MultiSet implements Collection {
private LinkedList l;

public MultiSet() {
l=new LinkedList();
}

public void add(Object o) {
l.add(o);
}

public boolean contains(Object o) {
return l.contains(o);
}

public boolean isEmpty() {
return l.isEmpty();
}
```

ESEMPIO DI IMPLEMENTAZIONE

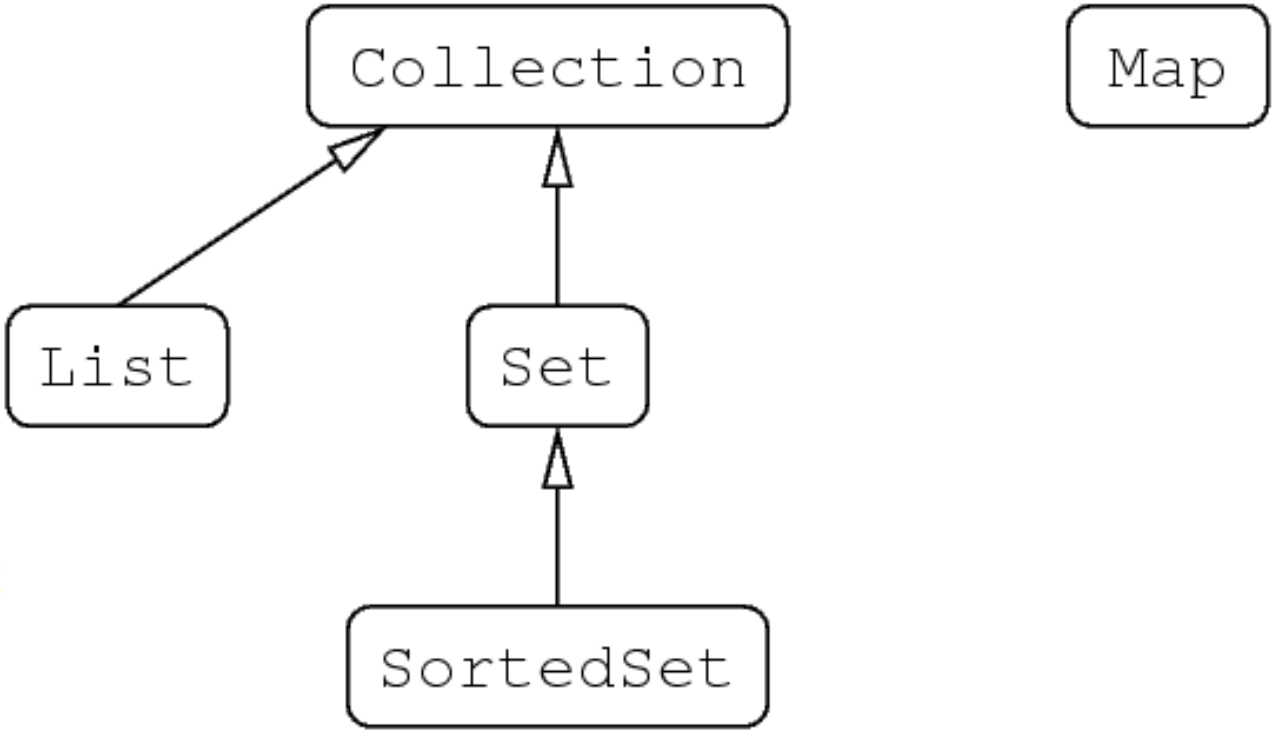
```
public Iterator iterator() {
return l.iterator();
}

public boolean remove(Object o) {
return l.remove(o);
}

public int size() {
return l.size();
}

// mancano altri metodi
}
```

IMPLEMENTAZIONE: COMMENTI

- Posso aggiungere un metodo `add(int i, Object p)`?
- nel linguaggio
  - quando si implementa una interfaccia: si devono implementare tutti i metodi, **più tutti quelli che voglio**
- nel contratto
  - un MultiSet deve rappresentare insiemi non ordinati; quindi, un metodo add(i, o) **è concettualmente sbagliato**
    - ma dal punto di vista sintattico potrei implementarlo senza ricevere errori

GERARCHIA DELLE INTERFACCE

- Le interfacce del *collection framework* sono organizzate cosí:



GERARCHIA DELLE INTERFACCE

- Il loro significato:
  - *Collection*
    - un multi-insieme non ordinato
  - *List*
    - una sequenza ordinata di elementi
  - *Set*
    - un insieme (non ordinato) di elementi
  - *SortedSet*
    - un insieme ordinato per compareTo
  - *Map*
    - un tabella (vedremo dopo cosa sono)

DIFFERENZA TRA LIST E SORTEDSET

- List
  - gli elementi sono in una sequenza qualsiasi
- SortedSet
  - gli elementi sono ordinati secondo l'ordinamento specificato da compareTo
  - **Attenzione**! Nell'interfaccia SortedSet non esiste il metodo *get(int index)*
- Gli iteratori di SortedSet scandiscono l'insieme *in ordine crescente*

ESEMPIO DI SORTEDSET

- La classe TreeSet implementa l'interfaccia SortedSet
- Esempio: inserisco alcuni interi e poi li stampo

ESEMPIO DI SORTEDSET

```
public static void main(String args[]) {
    SortedSet s=new TreeSet();

    s.add(new Integer(21));
    s.add(new Integer(0));
    s.add(new Integer(4));

    Iterator i=s.iterator();

    while(i.hasNext())
      System.out.println(i.next());
  }
```

ESEMPIO DI SORTEDSET

- Viene stampato:
- 0
  4
  21
- Gli iteratori scandiscono l'insieme in ordine
- L'ordinamento che si usa è quello dato da *compareTo*

ORDINAMENTO CON COMPARATORE

- Si puè creare un oggetto TreeSet che non usa l'ordinamento dato da compareTo
  - si crea una classe che implementa l'interfaccia *Comparable*
  - questa classe deve contenere un metodo (dinamico) *int compare(Object o1, Object o2)*
  - quando si crea l'oggetto *TreeSet*, si specifica un oggetto (qualsiasi) di questa classe
- Gli elementi sono nell'ordine dato da compare di questa classe, e non più nell'ordine di *compareTo* degli oggetti dell'insieme

ESEMPIO DI COMPARATORE

```
class Compara implements Comparator {
  public int compare(Object o1, Object o2) {
    int i, j;

    i=((Integer) o1).intValue();
    j=((Integer) o2).intValue();

    if(i<j)
      return 1;
    else if(i==j)
      return 0;
    else
      return -1;
  }
}
```

ESEMPIO DI COMPARATORE

- È l'ordine inverso di confronto fra interi
- Il programma di prima, modificato:

```
class Inverso {
  public static void main(String args[]) {
    Compara c=new Compara();
    SortedSet s=new TreeSet(c);

    s.add(new Integer(21));
    s.add(new Integer(0));
    s.add(new Integer(4));

    Iterator i=s.iterator();

    while(i.hasNext())
      System.out.println(i.next());
  }
}
```

ESEMPIO DI COMPARATORE

- Ora viene stampato:
- 21
  4
  0

LE MAPPE

- Servono a implementare "tavole con due colonne"
- Esempio: una tavola con un numero di matricola e il nome dello studente:

980123	"Ciccio"
879231	"Aldo"
843098	"Totti"
732108	"Aldo"

- Due studenti **possono avere lo stesso nome, ma non lo stesso numero di matricola**
- Gli oggetti mappa sono una generalizzazione

ASTRAZIONE

- Una tabella è caratterizzata da:
  - numero arbitrario di righe
  - in ogni riga ci sono due caselle
  - la prima casella contiene un identificatore univoco (nell' esempio: numero di matricola)
- Due caselle della prima riga non possono contenere lo stesso valore
- Le due colonne non hanno lo stesso significato:
  - prima colonna: un valore che permette di individuare univocamente la riga
  - seconda colonna: un valore qualsiasi
  - la prima colonna è la *"chiave di ricerca"*

MAP E HASHMAP

- *Map*
  - Interfaccia
- *HashMap*
  - un oggetto HashMap rappresenta una tabella
- *TreeMap*
  - una implementazione di Map che usa algoritmi diversi

KEY E VALUE

- *Key*
  - un oggetto della prima colonna
- *Value*
  - un oggetto della seconda colonna

METODI DI MAP E HASHMAP

- Metodi fondamentali:
  - *void put(Object key, Object value)*
    - inserisce la riga composta da key e value nella tabella
  - *boolean containsKey(Object key)*
    - vede se la tabella contiene una riga il cui primo elemento è key
  - *Object get(Object key)*
    - se la tabella contiene una riga con key in prima colonna, ritorna l'elemento in seconda

CREARE UNA TABELLA

- Questa tabella si può creare cosí:

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

   m=new HashMap();       // tabella vuota

   m.put(new Integer(980123),
         "Ciccio");

   m.put(new Integer(879231),
         "Aldo");

   m.put(new Integer(843098),
         "Totti");

   m.put(new Integer(732108),
         "Aldo");

}
```

980123	"Ciccio"
879231	"Aldo"
843098	"Totti"
732108	"Aldo"

- In memoria: la tabella contiene in effetti i riferimenti agli oggetti inseriti

PRESENZA ELEMENTO

- Per verificare se c'è uno studente con una certa matricola:

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

   // creazione tabella

   if(m.containsKey(new Integer(843040)))
     System.out.println("Matricola "+843040+" esistente");
   else
     System.out.println("Matricola "+843040+" non esistente");

   if(m.containsKey(new Integer(843098)))
     System.out.println("Matricola "+843098+" esistente");
   else
     System.out.println("Matricola "+843098+" non esistente");

  }
}
```

TROVARE UN ELEMENTO

- Trovare un elemento con una certa matricola:

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

    // creazione tabella
    // verifica presenza matricola 843098

    System.out.println("Lo studente con matricola "+843098+
                       " e': "+m.get(new Integer(843098)));
  }
}
```

GENERALIZZAZIONE

- Come key e value, al posto del numero di matricola e del nome, **posso usare oggetti qualsiasi**
- Esempio: numero di matricola e oggetto Studente

REALIZZAZIONE DI HASHSET

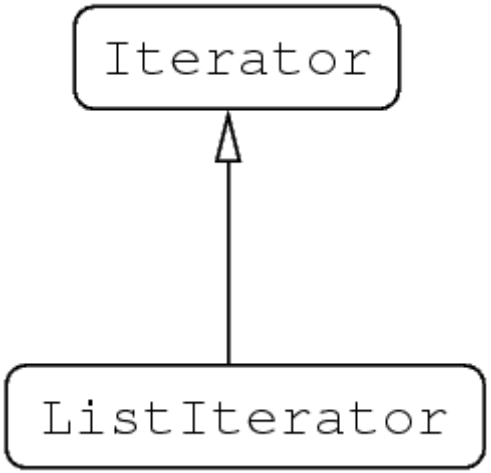
- In effetti gli HashSet sono realizzati usando una HashMap in cui si usa solo il campo key

GLI ITERATORI

- Un *interatore* è un oggetto che permette la scansione di un insieme
- Esiste un metodo della interfaccia *Collection* che restituisce un iteratore
- Quindi, **tutte le collezioni devono avere un iteratore associato**

GERARCHIA DEGLI ITERATORI

- Esistono due interfacce:



- *Iterator* : un cursore per fare la scansione di un insieme non necessariamente ordinato
- *ListIterator* : un iteratore di una classe che implementa List, ossia **un insieme ordinato**

METODI IN PIÙ DI LISTITERATOR

- Un oggetto *ListIterator* è un cursore per un oggetto che implementa List
- Consente di andare avanti e indietro
- Consente di inserire un elemento nella posizione corrente
- *hasPrevious()* ritorna True se esiste un elemento precedente nella lista

METODI IN PIÙ DI LISTITERATOR

- *Object previous()* ritorna l'elemento precedente della lista
- *void add(Object o)* inserisce un elemento
- *void set(Object o)* rimpiazza l'elemento con quello passato

METODI OPZIONALI

- Nelle interfacce, viene spesso specificato che un metodo è opzionale
- Naturalmente **deve** essere implementato, altrimenti viene dato errore
- linguaggio
  - una classe che implementa una interfaccia deve implementare tutti i suoi metodi

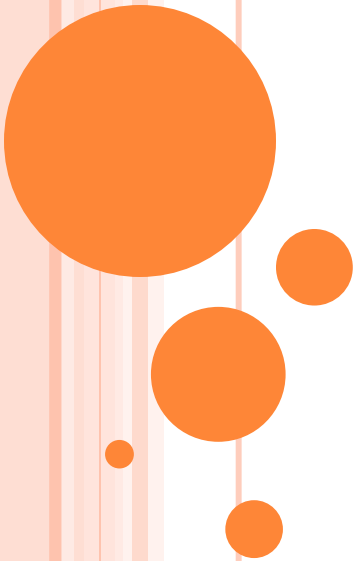
METODI OPZIONALI

- contratto
  - se un metodo è opzionale, la sua implementazione può semplicemente lanciare una eccezione:
  - *throw new UnsupportedOperationException("Operation not supported");*

ESEMPIO

- L'interfaccia *Collection* contiene un metodo *add* opzionale
- Ogni classe che implementa *Collection* lo deve contenere (altrimenti il compilatore dà errore)
- Dal momento che è opzionale, si può anche implementare in questo modo:

```
class Prova implements Collection {
   ...
  public boolean add(Object o) {
    throw new UnsupportedOperationException("Add operation not
supported");
  }
}
```



GLI HASHSET

Tipo predefinito che rappresenta insiemi di Object

COSA SUCCEDE SE...

- Posso mettere un riferimento a un Point in una variabile Object
- `Object o=new Point(12,3);`
  - *è quasi tutto come se l'oggetto fosse un Object*
- Unica eccezione:
  - *se invoco un metodo di Object che è ridefinito in Point, viene invocato il metodo definito in Point*

- Cosa succede se faccio queste cose:
- `o.x`
  - non funziona (x non è una componente di Object)
- `o.move(...)`
  - non funziona (il metodo move non è un metodo di Object)
- `o.equals(...)`
  - questo metodo sta sia in Object che in Point:
    viene invocato l'equals di Point
- Funziona tutto come se l'oggetto fosse un Object, tranne quando si invoca un metodo che sta sia in Object che in Point

- Dopo un cast `Point p=(Point) o`:
  variabile Point con dentro un riferimento a un Point:
  funziona tutto come al solito

COSA HANNO IN COMUNE

- Array, liste e insiemi
- Hanno in comune: una variabile rappresenta un insieme di oggetti

COSA HANNO DI DIVERSO

- Differenze
- operazioni elementari
  - negli array non si può inserire un elemento in mezzo: per farlo, devo creare un nuovo array e poi spostare gli elementi successivi
- Ordine
  - negli array e nella liste c'è un ordine degli elementi (primo elemento, secondo elemento, ecc) Negli insiemi non c'è nessun ordine

IL TIPO HASHSET

- Rappresenta insiemi non ordinati di elementi
- Non esiste un primo, secondo, ecc. elemento
- Creazione di un insieme vuoto:
  - HashSet h;
  - h=new HashSet();

METODI SUGLI INSIEMI

- add(Object o)
  - aggiunge un elemento (se non è già presente)
- contains(Object o)
  - vede se l'insieme contiene un elemento equals ad o
- remove(Object o)
  - rimuovo l'oggetto, se presente
- isEmpty()
  - vede se l'insieme è vuoto
- size()
  - ritorna il numero di elementi
- Nota: non c'è un ordine degli elementi
- Quindi, non si può dire ``inserisci in prima posizione'' oppure ``trova l'elemento in ultima posizione''

ESERCIZIO

- Creare un insieme che contiene i numeri interi da -5 a 5
- Considerazione
  - Dato che negli HashSet ci posso solo mettere oggetti, devo usare il tipo Integer

IMPLEMENTAZIONE

- Ciclo da -5 a 5
- Ogni valore lo metto in un oggetto Integer

```
import java.util.*;

class Valori {
  public static void main(String args[]) {
    HashSet s;
    s=new HashSet();

    int i;
    Integer v;

    for(i=-5; i<=5; i++) {
      v=new Integer(i);
      s.add(v);
    }

    System.out.println(s);
  }
}
```

COSA STAMPA?

- L'istruzione `println(s)` converte l'insieme in una stringa, e poi la stampa
- Viene stampato:
  - [5, -1, 4, -2, 3, -3, 2, -4, 1, -5, 0]
- Gli elementi non sono nell'ordine in cui li ho messi!

INSIEMI E ORDINE

- Per definizione, in un insieme non devo tenere conto dell'ordine in cui gli elementi vengono inseriti
  - {1, 2, 3} e {2, 1, 3} sono lo stesso insieme
- Sono due stringhe che rappresentano lo stesso insieme
- Il metodo `toString` degli insiemi dà *una* stringa fra quelle che rappresentano l'insieme

ESERCIZIO

- Scrivere un metodo statico che verifica se un insieme contiene due interi consecutivi fra -10 e 10

```
static boolean consecutivi(HashSet s) {
   ...
 }
```

SOLUZIONE

- Non abbiamo ancora visto come fare cicli su tutti gli elementi
- Facciamo un ciclo da -10 a 9
- A ogni passo, verifichiamo la presenza

```
static boolean consecutivi(HashSet s) {
   int i;
   Integer v, t;

   for(i=-10; i<=9; i++) {
     v=new Integer(i);
     t=new Integer(i+1);

     if(s.contains(v) && s.contains(t))
       return true;
   }

   return false;
 }
```

ESEMPIO

- Se viene passato l'insieme:
- [3, 1, 8, 0] Cosa restituisce il metodo?

- Restituisce true
- L'insieme [3, 1, 8, 0] contiene i numeri 0 e 1 che sono consecuitivi

COME FARE CICLI

- Si usa un meccanismo che esiste anche sulle liste
- Si crea un *iteratore*, che è un oggetto che serve a fare cicli sugli insiemi

ESEMPIO DI CICLO

- Questo ciclo stampa gli elementi dell'insieme s
- Si può fare la stessa cosa anche se s è una LinkedList

```
Iterator it;
   it=s.iterator();

   while(it.hasNext()) {
     v=(Integer) it.next();
     System.out.println(v);
   }
```

COSA FA UN ITERATORE

- Crea un ordine fra gli elementi dell'insieme, e permette di passare da un elemento a quello dopo
- it=s.iterator();
  - Crea un iteratore sull'insieme s
- it.next();
  - Trova il prossimo elemento dell'insieme
- it.hasNext()
  - Ritorna true se ci sono altri elementi nell'insieme

SCHEMA DI CICLO

- Per fare la stessa cosa su tutti gli elementi di una lista o di un array:

```
for(i=0; i<l.size(); i++) {
   elemento=l.get(i);

   opera su elemento
 }
```

- Per liste e insiemi, si può fare il ciclo con un iteratore:

```
Iterator it;
  it=s.iterator();

  while(it.hasNext()) {
    elemento=it.next();

    opera su elemento
}
```

- Per gli insiemi, non esiste ordinamento (non esiste l'indice di un element), quindi non c'è il metodo get (che ha come argomento un indice).
- L'unico modo di fare cicli per gli insiemi è quello di usare un iteratore

ESEMPIO

- Dato un insieme di punti, stampare solo quelli che stanno nel primo quadrante

```
static void primoQuadrante(HashSet s) {
   ...
}
```

ALGORITMO RISOLUTIVO

- *per ogni elemento dell'insieme*
  - *se e' nel primo quadrante, stampalo*

- Nelle liste, la parte "per ogni elemento" si può implementare con un iteratore oppure con il ciclo solito
- Per gli insiemi, posso solo usare l'iteratore

SOLUZIONE

- Creo un iteratore, e lo uso nel ciclo
- Per vedere le coordinate, devo fare il cast

```
static void primoQuadrante(HashSet s) {
   Point p;

   Iterator i;
   i=s.iterator();

   while(i.hasNext()) {
     p=(Point) i.next();

     if(p.x>=0 && p.y>=0)
       System.out.println(p);
   }
}
```

ALTRO ESERCIZIO

- Verificare se un insieme di interi contiene due numeri consecutivi

- `static boolean consecutivi(HashSet s) { ... }`

- Il metodo di prima lo faceva solo per interi da -10 a 10
- Ora lo voglio per tutti gli interi
- Non si può fare un ciclo su tutti i possibili valori interi

SOLUZIONE

- Si fa un ciclo su tutti gli elementi dell'insieme
- Per ogni elemento, se c'è anche quello più grande di uno, si ritorna true
- Se questo non è mai vero, si ritorna false

IMPLEMENTAZIONE DEL METODO

```
static boolean consecutivi(HashSet s) {
   int i;
   Integer v, t;

   Iterator it;
   it=s.iterator();

   while(it.hasNext()) {
     v=(Integer) it.next();
     i=v.intValue();
     t=new Integer(i+1);

     if(s.contains(t))
       return true;
   }

   return false;
 }
```

METODI DELL'INSIEME E DELL'ITERATORE

- Attenzione a non confondere *s* e *it*, e usare i metodi sull'oggetto sbagliato:
- `s.contains(...)`
  - la verifica di contenimento, cosí come l'inserimento, cancellazione, ecc. vanno fatte sull'insieme
- it.hasNext() e it.next()
  - questi due sono i metodi degli iteratori: sono quelli che permettono di fare il ciclo (trova l'elemento successivo, e verifica se ci sono elementi successivi)

ITERATORI E RIMOZIONE

- Se:
  - si crea un iteratore con it=s.iterator()
  - si modifica la lista con s.add oppure s.remove
- L'iteratore diventa non più valido (non si può più usare)
- Quindi, dopo la modifica non si può più invocare it.hasNext() oppure it.next()

ESERCIZIO

- Eliminare tutti gli elementi di valore pari da un insieme di interi di interi

SOLUZIONE SBAGLIATA

- Faccio il solito ciclo
- Quando trovo un elemento pari, lo elimino

```
static void eliminaPari(HashSet s) {
    Integer v;

    Iterator i;
    i=s.iterator();

    while(i.hasNext()) {
      v=(Integer) i.next();
      if(v.intValue()%2==0)
        s.remove(v);
    }
  }
```

MODIFICHE E ITERATORI

- Dopo una modifica a un insieme, tutti i suoi iteratori diventano non validi
- Invocare un metodo su un iteratore non valido produce un errore
- Nel metodo di prima, dopo aver fatto s.remove(v), si proseguiva il ciclo, per cui venivano fatte le invocazioni i.hasNext() e i.next(), che producono un errore

PRIMA SOLUZIONE CORRETTA

- Ogni volta che viene invocato s.iterator() viene creato un nuovo iteratore
- Questo iteratore riparte dall'inizio
- La soluzione: ogni volta che trovo un elemento pari, lo elimino e ricreo l'iteratore

```
static void eliminaPari(HashSet s) {
   Integer v;

   Iterator i;
   i=s.iterator();

   while(i.hasNext()) {
     v=(Integer) i.next();
     if(v.intValue()%2==0) {
       s.remove(v);
       i=s.iterator();
     }
   }
 }
```

SECONDA SOLUZIONE CORRETTA

- Richiede due metodi
  - verifica se ci sono elementi pari
  - elimina un elemento pari (es. il primo che si incontra)
- Intanto: implementare questi due metodi

VERIFICA SE CI SONO ELEMENTI PARI

- Solita cosa: ciclo su tutti gli elementi

```
static boolean contienePari(HashSet s) {
   Integer v;

   Iterator i;
   i=s.iterator();

   while(i.hasNext()) {
     v=(Integer) i.next();
     if(v.intValue()%2==0)
       return true;
   }

   return false;
 }
```

ELIMINAZIONE DI UN ELEMENTO PARI

- Solito ciclo: quando trovo un elemento pari lo elimino
- Dopo l'eliminazione, esco

```
static void eliminaUnPari(HashSet s) {
   Integer v;

   Iterator i;
   i=s.iterator();

   while(i.hasNext()) {
     v=(Integer) i.next();
     if(v.intValue()%2==0) {
       s.remove(v);
       break;
     }
   }
 }
```

- Attenzione: se invoco un metodo su i dopo una modifica, viene dato errore in esecuzione
- Verificare che non vengano fatte, per sbaglio, invocazioni di metodo sugli iteratori dopo una modifica

COSA CI FACCIO CON QUESTI DUE METODI?

- Domanda: ora che ho realizzato i due metodi, come faccio il metodo di eliminazione di tutti gli elementi?
- **Soluzione**
- Devo seguire questo algoritmo:
  - elimina un elemento pari
  - elimina un elemento pari
  - elimina un elemento pari
  - elimina un elemento pari ...
- Quando non ho più elementi pari, mi posso anche fermare

IMPLEMENTAZIONE

- Per eliminare tutti gli elementi pari da un insieme:
- Finchè ci sono elementi pari, ne elimino uno

```
static void eliminaPari(HashSet s) {
   while(contienePari(s))
     eliminaPrimoPari(s);
 }
```

METODO REMOVE DEGLI ITERATORI

- Elimina l'elemento corrente (l'ultimo che è stato ritornato da next)
- Dopo, l'iteratore si può ancora usare
- Note:
  - contrariamente al remove degli insiemi, non ha argomenti (l'elemento da rimuovere è l'ultimo ritornato da next)
  - invalida tutti gli altri iteratori dello stesso insieme, tranne l'iteratore su cui è invocato

ESEMPIO

```
HashSet unInsieme, unAltro;
…
Iterator a=unInsieme.iterator();
Iterator b=unInsieme.iterator();
Iterator c=unAltro.iterator();
…
Object o=a.next();
a.remove();
```

- Viene eliminato l'oggetto o dall'insieme *unInsieme*
- Ora a si può ancora usare, mentre b (e tutti gli eventuali altri iteratori di s) non sono più validi
- Iteratori di altri insiemi sono ancora validi; in questo caso c è ancora valido

INSIEMI DI OGGETTI ARBITRARI

- Si possono fare insiemi di `Point`, `String`, `Integer`, ecc.
- Se creo una nuova classe, come Studente, questa deve avere due metodi:
- equals
  - confronta due oggetti
- hashCode
  - dà un intero che viene usato dalla classe

METODO HASHCODE

- Vedremo poi perchè va messo
- Questo qui sotto funziona:

```
class NomeClasse {
  …
public int hashCode() {
  return 0;
 }
}
```

- Nella classe devono essere presenti sia equals che hashCode!

COSA SUCCEDE SE NON LO METTO?

- Non viene dato nessun errore, nè in compilazione, nè in esecuzione
- Cosa cambia?
- con hashCode
  - i metodi lavorano correttamente
- senza hashCode
  - i metodi lavorano come se equals fosse quello di Object
- Senza hashCode, il metodo contains verifica se c'è un oggetto == a quello passato, invece di verificare se c'è un oggetto equals

ATTENZIONE A EQUALS!

- Ci vuole *anche* il metodo equals

```
class Studente {
  String nome;
  int esami;
  double media;

  public String toString() {
    return "["+this.nome+" "+
           this.esami+" "+
           this.media+"]";
  }

  public boolean equals(Object o) {
    if(o==null)
      return false;

    if(this.getClass()!=o.getClass())
      return false;

    Studente s;
    s=(Studente) o;

    if(s.nome==null) {
      if(this.nome!=null)
        return false;
    }
    else
      if(!s.nome.equals(this.nome))
        return false;

    return ((s.esami==this.esami) &&
            (s.media==this.media));
  }

  public int hashCode() {
    return 0;
  }
}
```

UN ESEMPIO COMPLESSO

- Aggiungere a Studente un metodo statico `leggiStudente` che legge uno studente da un `BufferedReader` (ogni componente sta su una linea)
- Scrivere un metodo statico `leggiFile` della classe Studente che legge un intero file, il cui nome viene passato come argomento, e resituisce un oggetto HashSet che contiene gli studenti letti
  - Dato che serve un HashSet, devo avere i metodi equals e hashCode
  - Il metodo toString va messo sempre, perchè è comodo

```
class Studente {
  String nome;
  int esami;
  double media;

  public String toString() {
    return "["+this.nome+" "+
           this.esami+" "+
           this.media+"]";
  }

  public boolean equals(Object o) {
    if(o==null)
      return false;

    if(this.getClass()!=o.getClass())
      return false;

    Studente s;
    s=(Studente) o;

    return ((s.nome.equals(this.nome)) &&
            (s.esami==this.esami) &&
            (s.media==this.media));
  }

  public int hashCode() {
    return 0;
  }

  // altri metodi
}
```

LETTURA DI UNO STUDENTE DA FILE

- Il metodo prende come paramentro un BufferedReader
- Deve solo leggere tre linee, e convertirle se necessario
- Dato che uso i file, serve l'import, e devo fare throws IOException

```
import java.io.*;

class Studente {
  // componenti e metodi

  static Studente leggiStudente
    (BufferedReader b)
    throws IOException {
    Studente t;
    t=new Studente();

    String s;

    s=b.readLine();
    if(s==null)
      return null;
    t.nome=s;

    s=b.readLine();
    if(s==null)
      return null;
    t.esami=Integer.parseInt(s);

    s=b.readLine();
    if(s==null)
      return null;
    t.media=Double.parseDouble(s);

    return t;
  }
```

PERCHÈ TORNARE NULL?

- Il metodo leggiStudente ritorna null quando il file è finito
- Il metodo/programma che lo invoca, può vedere il valore di ritorno per capire se il file è finito
- Esempio: il metodo leggiFile (prox pagina) usa questo fatto per capire quando non ci sono più oggetti da leggere

LETTURA DI UN HASHSET

- Qui il parametro è il *nome* di un file
- Quindi, il FileReader e il BufferedReader vanno creati
- Dato che sono operazioni su file, bisogna mettere throws IOException
- Dato che serve il tipo HashSet, devo anche fare import java.util.*;

```
import java.io.*;
import java.util.*;

class Studente {
  // componenti e altri metodi

  static HashSet leggiFile(String nomefile)
    throws IOException {
    FileReader r=new FileReader(nomefile);
    BufferedReader b=new BufferedReader(r);

    HashSet h;
    h=new HashSet();

    Studente s;

    while(true) {
      s=leggiStudente(b);
      if(s==null)
        break;
      h.add(s);
    }

    return h;
  }
}
```

- Si tratta soltanto di leggere oggetti Studente da file, e metterli nell'insieme, fino a che non si legge un oggetto null

UN PROGRAMMA DI PROVA

- Leggere il file roma1.txt in un HashSet
- Stampare l'insieme (un oggetto per linea)
- Contare quanti studenti hanno dato meno di cinque esami
- Stampare i dati dello studente Totti, se si trova nell'insieme

SOLUZIONE

- Anche se non è richiesto: conviene fare un metodo per ognuna delle cose da fare
- Metodo di stampa:

```
static void stampa(HashSet s) {
   Iterator i;
   i=s.iterator();

   while(i.hasNext())
     System.out.println(i.next());
 }
```

TROVARE STUDENTI CON POCHI ESAMI

- Metodo che trova gli studenti con meno di un certo numero di esami:

```
static int contaMeno(HashSet s, int quanti) {
   Iterator i;
   i=s.iterator();

   int conta=0;

   Studente q;

   while(i.hasNext()) {
     q=(Studente) i.next();
     if(q.esami<quanti)
       conta++;
   }

   return conta;
 }
```

- Perchè il parametro quanti?
- È bene cercare di fare metodi generali, perchè si possono riusare
- In questo caso: l'unica cosa che cambiava era che: if(q.esami<5) diventa if(q.esami<quanti)
- Se la differenza è piccola conviene cercare di scrivere metodi più generali
- Se la generalità comporta una complicazione, meglio il metodo non generale

STAMPA I DATI DI TOTTI

- Faccio un ciclo di scansione
- Se il nome dello studente è Totti, stampo tutto lo studente

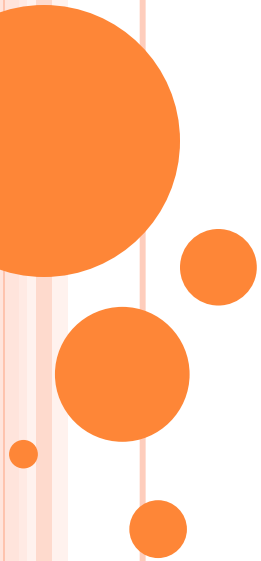
```
 static void stampaStudente(HashSet s, String nome)
{
    Iterator i;
    i=s.iterator();

    Studente q;

    while(i.hasNext()) {
      q=(Studente) i.next();
      if(q.nome.equals(nome))
        System.out.println(q);
    }
  }
```

IL METODO MAIN

- Il programma completo è fatto cosí

```
import java.util.*;
import java.io.*;

class ProvaStudente {
  // altri metodi

  public static void main(String args[])
    throws IOException {
    HashSet tutti;
    tutti=Studente.leggiFile("roma1.txt");

    stampa(tutti);

    System.out.println(contaMeno(tutti, 5));

    stampaStudente(tutti, "Totti");
  }
```

- Notare che il metodo main invoca il metodo Studente.leggiFile che opera con file.
- Quindi, devo fare throws IOException
- In generale: se si presenta l'errore di compilazione unreported exception..., basta aggiungere throws



LE TAVOLE HASH

È il modo con cui sono realizzati gli HashSet

Si usano quando serve accesso rapido sia in lettura che in scrittura su un insieme non ordinato

PRINCIPIO BASE

- Gli array hanno le caratteristiche che servono:
  - lettura di un elemento: vett[i]
  - scrittura di un elemento: vett[i]=a
- Sono tutte e due operazioni dirette
  (non richiedono nessuna ricerca)
- Cerco di usare un vettore per rappresentare un insieme

VETTORE CARATTERISTICO DI UN INSIEME

- Un insieme di *numeri interi* che vanno da 0 a 100 si può rappresentare usando un vettore di cento elementi booleani
- Se v è il vettore che rappresenta l'insieme, le operazioni si realizzano cosí:
- Lettura
  - l'intero x si trova nell'insieme se e solo se v[x] è true
- Scrittura
  - per inserire l'intero x, si fa v[x]=true
  - per togliere, si fa v[x]=false

INSIEMI GENERICI

- Limiti della rappresentazione dei vettori caratteristici:
  - grandezza del vettore=numero di tutti gli elementi possibili
  - si rappresentano solo interi
- Si possono superare tutti e due grazie agli hashCode

IL METODO HASHCODE

- È un metodo che ritorna un intero dato un oggetto
- Dato un oggetto o, il suo intero corrispondente è o.hashCode()
- Per il momento ci basta sapere questo

MODIFICA DELLA RAPPRESENTAZIONE

- Al posto del vettore di booleani, usiamo un vettore di Object
  - il codice hash di un oggetto identifica una posizione nel vettore
  - se l'oggetto c'è, sta in quella posizione

```
class Tavola {
  private Object t[];

  public Tavola() {
    t=new Object[...];
    // inizializzazione automatica a null
  }

  boolean add(Object o) {
    int c=o.hashCode();

    if(t[c]!=null)
      return false;

    t[c]=o;
    return true;
  }

  ...
```

METODI CONTAINS E REMOVE

```
class Tavola {
  ...

  boolean remove(Object o) {
    boolean p;
    int c=o.hashCode();

    p=(t[c]!=null);

    t[c]=null;

    return p;
  }

  boolean contains(Object o) {
    int c=o.hashCode();

    if(o==null)
      return false;

    if(t[c]==null)
      return false;

    return t[c].equals(o);
  }
}
```

DUE PROBLEMI

- due oggetti possono avere lo stesso hashCode
- il vettore deve avere un elemento per ogni possibile numero intero
  Dato che si usano 32 bit per rappresentare un intero, servirebbe un vettore con 4294967296 elementi

- Del primo problema ci occupiamo dopo

DIMENSIONE DEL VETTORE

- Scelgo una dimensione qualsiasi (per esempio, 100)
- Quando l'hashCode vale c, uso c%100 per indicizzare l'array

```
class Tavola {
  private final int size=100;
  private Object t[];

  public Tavola() {
    t=new Object[size];
  }

  boolean add(Object o) {
    int c=o.hashCode()%size;

    if(t[c]!=null)
      return false;

    t[c]=o;
    return true;
  }
```

```
 boolean remove(Object o) {
    boolean p;
    int c=o.hashCode()%size;

    p=(t[c]!=null);

    t[c]=null;

    return p;
  }

  boolean contains(Object o) {
    int c=o.hashCode()%size;

    if(o==null)
      return false;

    if(t[c]==null)
      return false;

    return t[c].equals(o);
  }
}
```

OGGETTI CON LO STESSO CODICE HASH

- Soluzione banale: quando vado a inserire un oggetto ma la sua posizione è già occupata, lo metto invece in una lista
- Casella di un oggetto o: casella nella posizione o.hashCode()%100
- Dati
  - un array e una lista
- Inserimento
  - se la casella dell'oggetto è vuota, ci metto l'oggetto
  - altrimenti, metto l'oggetto nella lista
- ricerca
  - se la casella dell'oggetto contiene null, l'oggetto non c'è
  - se contiene un oggetto equals, ritorna true
  - se contiene un oggetto che non è uguale, allora bisogna andare a vedere nella lista!
- Cancellazione
  - stessa cosa; se l'oggetto sta nell'array, occorre anche spostare un eventuale oggetto dalla lista

DATI

- Un vettore e una lista, più la dimensione del vettore

```
import java.util.*;

class Tavola {
  private final int size=100;
  private Object t[];
  private LinkedList l;

  public Tavola() {
    t=new Object[size];
    l=new LinkedList();
  }

  ...
}
```

- La lista si chiama lista di trabocco

ASSUNZIONI

- Dato un oggetto o, la sua posizione naturale è:
  - posizione naturale di un oggetto o = casella di indice o.hashCode()%size del vettore
- Le assunzioni che faccio e rispetto sono:
  - un oggetto si trova preferibilmente nella sua posizione naturale
  - se la posizione naturale è occupata, allora l'oggetto sta da qualche altra parte
- Nel nostro caso, "da qualche altra parte" significa nella lista

INSERIMENTO

- Facile: se la posizione naturale è libera, ci metto l'oggetto
- Altrimenti, lo metto nella lista
- Prima devo controllare che non sia presente

```
boolean add(Object o) {
   int c=o.hashCode()%size;

   if(t[c]!=null)
     if(l.contains(o))
       return false;
     else
       return l.add(o);

   t[c]=o;
   return true;
 }
```

RICERCA

- Devo considerare tutte e due le assunzioni:
  - se l'oggetto lo trovo nella sua posizione naturale, allora c'è
  - se nella sua posizione naturale c'è un altro oggetto, allora devo guardare la lista

```
boolean contains(Object o) {
   int c=o.hashCode()%size;

   if(t[c]==null)
     return false;

   if(t[c].equals(o))
     return true;

   return l.contains(o);
 }
```

CANCELLAZIONE

- Faccio prima una ricerca
- Vedo se l'oggetto sta nella posizione naturale, e se è vuota guardo la lista
- Se trovo l'oggetto nella posizione naturale, non basta cancellarlo
- Se lo faccio, potrei avere un altro oggetto nella lista che ora non si trova nella sua posizione naturale anche se questa è libera!
- Sarebbe un oggetto che non trovo quando faccio `contains`

ELIMINAZIONE DALLA POSIZIONE NATURALE

- Esempio: si supponga che gli oggetti new Integer(50) e new Integer(150) abbiano entrambi posizione naturale 50
- Si inserisce prima 50 e poi 150
- Ora 50 sta nell'array e 150 nella lista
- Si elimina 50 mettendo null nell'array
- Se ora eseguo contains(new Integer(150)), mi ritorna false
- Infatti, il metodo contains guarda nella posizione naturale, trova null e ritorna false

ELIMINAZIONE DALLA POSIZIONE NATURALE

- Due soluzioni:
  - modifico contains in modo che cerca comunque nella lista
  - modifico remove
- La prima è inefficiente: quasi tutte le volte si deve andare a guardare nella lista
- Usiamo la seconda soluzione

ELIMINAZIONE: MODIFICA DELLA REMOVE

- L'assunzione che viene usata da contains è che un elemento sta nella lista solo se la suo posizione naturale è occupata
- Questo è lo stesso di: un elemento non può stare nella lista se la sua posizione naturale è libera
- Quando si elimina un elemento dalla sua posizione naturale, basta prendere dalla lista uno qualsiasi degli altri elementi con quella posizione naturale (se esiste) e metterlo nella posizione naturale
- Questo rende vera l'assunzione: se ci sono elementi nella lista, la loro posizione naturale è occupata

```
 boolean remove(Object o) {
    boolean p;
    int c=o.hashCode()%size;

    if(o==null)
      return false;

    if(t[c]==null)
      return false;

    if(!t[c].equals(o))
      return l.remove(o);

// trovato oggetto: guarda la lista
    Iterator i=l.iterator();
    while(i.hasNext()) {
      Object b=i.next();
      if(b.hashCode()%size==c) {
        t[c]=b;
        i.remove();
        return true;
      }
    }

    t[c]=null;
    return true;
  }
```

IL METODO HASHCODE

- Serve per trovare la posizione naturale di un oggetto in una tavola hash.
- Vincolo: se due oggetti sono equals, devono avere lo stesso hashCode
- La versione qui sotto rispetta questa specifica:
  - `public int hashCode() { return 0; }`
- Tutte le operazioni funzionano perchè la specifica è rispettata

EFFICIENZA DELLE TAVOLE HASH

- Se non vado mai nella lista di trabocco, ogni operazione ha costo uno (accesso al vettore)
- Se tutti gli oggetti hanno lo stesso codice hash, allora tutti gli oggetti tranne uno stanno nella lista
- L'efficienza delle operazioni è maggiore se riesco a non usare la lista di trabocco
- Il metodo hashCode dovrebbe funzionare in modo tale da restituire interi possibilmente diversi per gli oggetti diversi che uso
- Questo non può essere garantito; basta che due oggetti diversi che uso abbiano "spesso" codici diversi

IL METODO HASHCODE, RIVISTO

- Vogliamo avere risultato diverso se i campi sono diversi

```
class Studente {
  String name;
  int anno;

  public int hashCode() {
    int res=0;

    res+=name.hashCode();
    res+=anno;

    return res;
  }
}
```

- Prima versione: faccio la somma delle componenti intere e degli hashcode delle componenti oggetto

PROBLEMA DELLA PRIMA VERSIONE

- Supponiamo di avere un insieme di Point
- Non è improbabile che il punto (1,2) e (2,1) stiano nello stesso insieme
- Hanno però lo stesso hashCode
- Altro esempio: le persone con nome e cognome Bruno, Marco e Marco, Bruno hanno lo stesso hashCode se nome e cognome sono due componenti separate

VERSIONE MIGLIORATA DI HASHCODE

- Soluzione: ogni componente viene moltiplicata per un valore diverso

```
class Studente {
  String name;
  int anno;

  public int hashCode() {
    int res=3;

    res=5*res+name.hashCode();
    res=5*res+anno;

    return res;
  }
}
```

- L'ultima componente è moltiplicata per 1, la penultima per 5, la terz'ultima per 5*5, ecc.

VARIANTI DELLE TAVOLE HASH

- I concetti comuni delle tavole hash sono:
  - per ogni oggetto, c'è una posizione naturale in un vettore
  - se questa posizione è occupata, l'oggetto sta "da qualche altra parte"
- Ci sono versioni diverse in cui cambia il "qualche altra parte"

VERSIONE SENZA LISTA

- Se la posizione naturale di un oggetto è occupata, lo metto nella successiva posizione dell'array
- Variante: invece della posizione successiva uso posizione+incremento, dove incremento è un numero primo rispetto alla dimensione dell'array

SENZA LISTA: INSERIMENTO E CANCELLAZIONE

- Inserimento
  - nella posizione naturale se libera, altrimenti si cerca la prima posizione libera
- Ricerca
  - nella posizione naturale; se c'è un oggetto ma non è uguale, si passa alla posizione successiva e si ripete ricorsivamente

SENZA LISTA: CANCELLAZIONE

- È complicata
- Esempio: (il numero e' la posizione naturale di un oggetto)
  - prima: *{null null 2a 2b 2c 3d 3e null null}*
- Cancellare 2b: non posso mettere solo null nella posizione 3, altrimenti otterrei:
  - dopo: *{null null 2a null 2c 3d 3e null null}*
- Se ora cerco 2c oppure 3d, ecc. non li trovo

- Occorre spostare tutti gli elementi che seguono fino al primo elemento che è null
  - dopo: *{null null 2a 2c 3d 3e null null null}*
- In questi spostamenti, gli elementi non possono risalire prima della loro posizione naturale:
  - prima: *{null null 2a 2b 2c 3d 3e 7f 3g}*
  - non va:*{null null 2a null 2c 3d 3e 7f null}*
  - dopo: *{null null 2a 2c 3d 3e 3g 7f null}*
- Se non si rispettano queste regole, la cancellazione non è più coerente con la ricerca

PIÙ LISTE DI TRABOCCO

- Idea: per ogni elemento dell'array, ho una lista di trabocco
- Ho quindi un array di oggetti e un array di liste; i due array hanno la stessa dimensione
- L'elemento sta nella posizione naturale oppure nella lista di trabocco della posizione naturale

BUCKETS

- Realizzo solo l'array di liste
- Il principio è che l'accesso al primo elemento di lista è comunque facile come accedere a un elemento di un array
- Ognuna delle liste si chiama *bucket*

```
import java.util.*;

class Tavola {
  private final int size=100;
  private LinkedList l[];

  public Tavola() {
    int i;

    l=new LinkedList[size];
    for(i=0; i<l.length; i++)
      l[i]=new LinkedList();

  }

  boolean add(Object o) {
    int c=o.hashCode()%size;

    if(l[c].contains(o))
      return false;

    return l[c].add(o);
  }

  boolean contains(Object o) {
    int c=o.hashCode()%size;

    return l[c].contains(o);
  }

  boolean remove(Object o) {
    int c=o.hashCode()%size;

    return l[c].remove(o);
  }
}
```

REALIZZAZIONE: NOTA

- Si poteva anche lasciare l'array di liste vuoto
- La lista l[c] veniva creato solo quando si andava a inserire un elemento che aveva posizione naturale c
- Bisogna fare un controllo *l[c]==null* sia su remove che su contains

MATERIALE

- http://www.dis.uniroma1.it/~liberato/laboratorio/collection/collection.html