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`

Soluzione

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?

Risposta

`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`

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

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

Si può fare la stessa cosa anche se `s` è una `LinkedList`

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);
  }
}
```

Perchè non funziona?

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

---

Soluzione

Dato che serve un `HashSet`, devo avere i metodi `equals` e `hashCode`:

```
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
}
```

Il metodo `toString` va messo sempre, perchè è comodo

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;
  }
```

Sembra lungo, ma fa solo questo: legge tre linee, per ogni linea controlla se vale `null` (e in questo caso, ritorna `null`), la converte se necessario, e la mette nell'oggetto creato

---

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);
    }
  }
```

Notare anche qui: fare un metodo generale comporta solo una piccola complicazione (un parametro in più, e poi la modifica della condizione dell'`if`)

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`