Università degli Studi di Roma “La Sapienza”
Facoltà di Ingegneria – Corso di Laurea in Ingegneria Gestionale

Corso di Progettazione del Software

Proff. Toni Mancini e Monica Scannapieco
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”

S.JOO.4 – Java: uguaglianza superficiale e profonda

versione del February 2, 2008

Uguaglianza fra valori di un tipo base

Se vogliamo mettere a confronto due valori di un tipo base, usiamo l'operatore di uguaglianza '=='.

Ad esempio:

```
int a = 4, b = 4;
if (a == b) // verifica uguaglianza fra VALORI
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");
```

Uguaglianza fra oggetti

Quando confrontiamo due oggetti dobbiamo chiarire che tipo di uguaglianza vogliamo utilizzare:

- **Uguaglianza superficiale**: verifica se due riferimenti ad un oggetto sono uguali, cioè denotano lo stesso oggetto;
- **Uguaglianza profonda**: verifica se le informazioni (rilevanti) contenute nei due oggetti sono uguali.

Uguaglianza fra oggetti (cont.)

```
class C {
  private int x, y;
  public C(int x, int y) {
    this.x = x; this.y = y;
  }
}
// ...
    C c1 = new C(4,5);
    C c2 = new C(4,5);
```

Nota: `c1` e `c2` ...

- ... non sono uguali superficialmente
- ... sono uguali profondamente

Uguaglianza superficiale

Se usiamo '==' per mettere a confronto **due oggetti**, stiamo verificandone l'uguaglianza superficiale.

Ad esempio:

```
class C {
  private int x, y;
  public C(int x, int y) {this.x = x; this.y = y;}
}
// ...
    C c1 = new C(4,5), c2 = new C(4,5);
    if (c1 == c2)
      System.out.println("Uguali!");
    else
      System.out.println("Diversi!");
```

Uguaglianza superficiale (cont.)

Viene eseguito il ramo `else` (`"Diversi!"`).

Infatti, '==' effettua un confronto fra i valori dei riferimenti, ovvero fra i due indirizzi di memoria in cui si trovano gli oggetti.

Riassumendo, diciamo che:

1. '==' verifica l'uguaglianza superficiale,
2. gli oggetti `c1` e `c2` **non sono uguali superficialmente**.

Uguaglianza fra oggetti: funz. equals()

La funzione `public boolean equals(Object)` definita in `Object` ha lo scopo di verificare l'uguaglianza fra oggetti.

`equals()`, come tutte le funzioni definite in `Object`, è **ereditata** da ogni classe (standard, o definita dal programmatore), e se non ridefinita, si comporta come l'operatore '=='.

Pertanto, anche nel seguente esempio viene eseguito il ramo `else` (`"Diversi!"`).

```
class C {
  int x, y;
  public C(int x, int y) {this.x = x; this.y = y;}
}
// ...
    C c1 = new C(4,5), c2 = new C(4,5);
    if (c1.equals(c2))
      System.out.println("Uguali!");
    else
      System.out.println("Diversi!");
```

Uguagl. profonda: overriding di equals()

È tuttavia possibile **ridefinire** il significato della funzione `equals()`, facendone **overriding**, in modo tale da verificare l'**uguaglianza profonda** fra oggetti.

Per fare ciò dobbiamo ridefinire la funzione `equals()` come illustrato nel seguente esempio:

```
class B {
  private int x, y;

  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      B b = (B)o;
      return (x == b.x) && (y == b.y);
    }
    else return false;
  }
}
```

Analisi critica dell'overriding di equals()

Alcuni commenti sulla funzione `equals()` ridefinita per la classe `B`:

- `public boolean equals(Object o) {` la funzione deve avere come parametro un riferimento di tipo `Object` perchè stiamo facendo **overriding** della funzione `equals()` ereditata dalla classe `Object`.
- `if (o != null ...`
  dobbiamo essere sicuri che il riferimento o passato alla funzione non sia `null`, altrimenti gli oggetti sono banalmente diversi, visto che l'oggetto passato alla funzione non è un oggetto;
- `... && getClass().equals(o.getClass())`
  dobbiamo essere sicuri che o si riferisca ad un oggetto della stessa classe dell'oggetto di invocazione (`B`, nell'esempio), altrimenti i due oggetti sono istanze di classi diverse e quindi sono ancora una volta banalmente diversi;
- `B b = (B)o;`
  se la condizione logica dell'`if` risulta vera, allora facendo un **cast** denotiamo l'oggetto passato alla funzione attraverso un riferimento del tipo dell'oggetto di invocazione (`B`, nell'esempio) invece che attraverso un riferimento generico di tipo `Object`; in questo modo potremo accedere ai campi specifici della classe di interesse (`B`, nell'esempio)
- `return (x == b.x) && (y == b.y)`
  a questo punto possiamo finalmente verificare l'uguaglianza tra i singoli campi della classe

Overriding, non overloding, di equals()

Si noti che si deve fare **overriding** di `equals()` e **non overloading**. Altrimenti si possono avere risultati controintuitivi.
Cosa fa questo programma?

```
// File Codice/J2/Esercizio12.java

class B {
  private int x, y;
  public B(int a, int b) {
    x = a; y = b;
  }
  public boolean equals(B b) { // OVERLOADING, NON OVERRIDING
    if (b != null)
      return (b.x == x) && (b.y == y);
    else return false;
  }
}

public class Esercizio12 {
  static void stampaUguali(Object o1, Object o2) {
    if (o1.equals(o2))
      System.out.println("I DUE OGGETTI SONO UGUALI");
    else
      System.out.println("I DUE OGGETTI SONO DIVERSI");
  }

  public static void main(String[] args) {
    B b1 = new B(10,20);
    B b2 = new B(10,20);

    if (b1.equals(b2))
      System.out.println("I DUE OGGETTI SONO UGUALI");
```

Uguaglianza fra oggetti: profonda (cont.)

Riassumendo, se desideriamo che per una classe `B` si possa verificare l'uguaglianza profonda fra oggetti, allora:

server: il **progettista** di `B` deve effettuare l'overriding della funzione `equals()`, secondo le regole viste in precedenza;

client: il **cliente** di `B` deve effettuare il confronto fra oggetti usando `equals()`.

```
B b1 = new B(), b2 = new B();
b1.x = 4; b1.y = 5;
b2.x = 4; b2.y = 5;
if (b1.equals(b2))
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");
```

Uguaglianza: classe String

In `String` la funzione `equals()` è ridefinita in maniera tale da realizzare l'uguaglianza profonda.

```
String s1 = new String("ciao");
String s2 = new String("ciao");

if (s1 == s2)
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");

if (s1.equals(s2))
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");
```

Esercizio 13: uguaglianza

Progettare tre classi:

`Punto`: vedi esercizio proposto in precedenza;

`Segmento`: vedi esercizio proposto in precedenza;

`Valuta`: per la rappresentazione di una quantità di denaro, come aggregato di due valori di tipo intero (unità e centesimi) ed una `String` (nome della valuta).

Per tali classi, ridefinire il significato della funzione `equals()`, facendo in maniera tale che verifichi l'uguaglianza profonda fra oggetti.

Uguaglianza profonda in classi derivate

Se desideriamo specializzare il comportamento dell'uguaglianza per una classe `D` derivata da `B`, si può fare overriding di `equals()` secondo il seguente schema semplificato:

```
public class D extends B {
  protected int z;
  public boolean equals(Object ogg) {
    if (super.equals(ogg)) {
      D d = (D)ogg;
      // test d'uguaglianza campi dati specifici di D
      return z == d.z;
    }
    else return false;
  }
}
```

- `D.equals()` delega a `super.equals()` (cioè `B.equals()`) alcuni controlli (**riuso**):
  - che il parametro attuale non sia `null`;
  - che l'oggetto di invocazione ed il parametro attuale siano della stessa classe;
  - che l'oggetto di invocazione ed il parametro attuale coincidano nei campi della classe base.
- `D.equals()` si occupa solamente del controllo dei campi dati specifici di `D` (cioè di `z`).

Esercizio 14: cosa fa questo programma?

```
class B { // ... la solita

class D extends B {
  protected int z;
  public D(int a, int b, int c) {//...
  public boolean equals(Object ogg) {
    if (super.equals(ogg)) {
      D d = (D)ogg;
      return z == d.z;
    }
    else return false;
  }
}
// ...
    D d = new D(4,5,6);
    E e = new E(4,5,6);

    if (d.equals(e))
      System.out.println("I DUE OGGETTI SONO UGUALI");
    else
      System.out.println("I DUE OGGETTI SONO DIVERSI");
```

```
class E extends B {
  protected int z;
  public E(int a, int b, int c){//...
  public boolean equals(Object ogg) {
   if (super.equals(ogg)) {
     E e = (E)ogg;
     return z == e.z;
   }
   else return false;
  }
}
```

Uguagl. profonda: hashCode()

Il controllo di uguaglianza profonda mediante l'esecuzione del metodo `equals()` (ridefinito) può essere abbastanza costoso.

Alcune classi, come quelle che realizzano tipi di dati come insiemi ecc., hanno la necessità di effettuare tali controlli spesso (ad esempio, nel realizzare il tipo di dato Insieme, ad ogni inserimento dobbiamo verificare che l'elemento da inserire non sia "equal" di uno già esistente.

A tale scopo, la classe `Object` mette a disposizione un'ulteriore funzione: `public int hashCode()`, con l'intento di fornire un metodo efficiente per controllare che due oggetti non siano "equal".

In particolare, il metodo `hashCode()`, che ritorna un intero, rispetta il seguente contratto:

Dati due oggetti `o1` e `o2`: **se** `o1.hashCode() != o2.hashCode()` **allora** sicuramente `o1.equals(o2) == false`.

Al contrario, se `o1.hashCode() == o2.hashCode()`, non si può inferire nulla: i due oggetti possono essere o meno equal. È necessario invocare `o1.equals(o2)` per assicurarsene.

La funzione hashCode() (2)

Lo scopo della funzione `Object.hashCode()` (che può essere ridefinita nelle sottoclassi allo stesso modo di `equals()`) è quello di fornire un controllo efficiente anche se parziale della disuguaglianza profonda di due oggetti.

Il metodo `hashCode()` è, come `equals()` già definito nella classe Object, ed è di solito implementato di modo che ritorni l'indirizzo di memoria dell'oggetto di invocazione. Questa implementazione è consistente con il contratto e con l'implementazione del metodo `equals()` della classe Object:

Infatti, dati riferimenti `o1` e `o2` ad oggetti di classe (più specifica) Object:

1. `o1.equals(o2) == true` se e solo se `o1 == o2`, ovvero se e solo se `o1` e `o2` si riferiscono allo stesso oggetto in memoria;
2. Quindi, se `o1.hashCode() != o2.hashCode()`, si ha sicuramente che `o1.equals(o2) == false`, visto che `Object.hashCode()` ritorna l'indirizzo in memoria dell'oggetto di invocazione (indirizzi diversi denotano oggetti distinti!)

Overriding di hashCode() (1)

Dal comportamento di `Object.hashCode()` ne consegue che, per rispettarne il contratto, ovvero:

Se `o1.hashCode() != o2.hashCode()` **allora** `o1.equals(o2) == false`

ogni volta che, per una classe, si ridefinisce il metodo `equals()`, va ridefinito anche il metodo `hashCode()`, di modo che ritorni **lo stesso intero** se invocato su due oggetti equal tra loro.

Se non si ridefinisce `hashCode()` si può incorrere nel seguente comportamento errato di un cliente che lo utilizza.

Esempio: equals() vs. hashCode()

Esempio di overriding di `equals()` ma non di `hashCode()`:

```
public class Punto {
   private double x, y, z;
   public Punto(double xx, yy, zz) { x=xx; y=yy; z=zz; }
   ...
   public Object equals(Object o) {
      if (o == null) return false;
      if (!this.getClass().equals(o.getClass())) return false;
      return x=o.x && y=o.y && z=o.z;
   }
   // Non effettuo overriding di hashCode()
}
```

Supponiamo di avere due oggetti `p1` e `p2` di classe Punto creati nel modo seguente:

```
Punto p1 = new Punto(3, 3, 3);
Punto p2 = new Punto(3, 3, 3);
```

Si noti che `p1.equals(p2) == true`.

Esempio: equals() vs. hashCode() (cont.)

Supponiamo ora che un modulo cliente (ad esempio il metodo `aggiungi(Punto p)` di una classe che realizza il tipo di dato Insieme(Punto)) utilizzi `hashCode()` per controllare efficientemente che `p1` e `p2` non siano “equal”. Si avrebbe che:

- Dato che la classe Punto non ridefinisce `hashCode()`, il comportamento di tale funzione è quello ereditato dalla classe Object;
- Pertanto `p1.hashCode()` e `p2.hashCode()` ritornano interi diversi (rappresentanti gli indirizzi di memoria dei due oggetti, che sono distinti);
- Quindi `p1.hashCode() != p2.hashCode()` nonostante `p1.equals(p2) == true`.

Dal fatto che gli hash-code di `p1` e `p2` sono diversi, il modulo cliente concluderà (v. il contratto di hashCode()) che `p1` e `p2` **non** sono equal.

Overriding di hashCode() (2)

Per evitare tali malfunzionamenti, ogni volta che una classe ridefinisce il metodo `equals()` deve ridefinire anche il metodo `hashCode()`, di modo che ritorni lo stesso intero quando invocato su oggetti che sono equal tra loro.

Osservazione. Un banale (ma corretto!) overriding del metodo `hashCode()` è il seguente:

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

Questa implementazione rispetta il contratto di `hashCode()`:

Se due oggetti hanno diverso hash-code, **allora** non sono equal

(si noti che l'uguaglianza tra due hash-code non permette di inferire nulla sulla uguaglianza profonda dei due oggetti!)

Tuttavia, tale implementazione non è affatto utile: di fatto stiamo impedendo ad un cliente di utilizzare hashCode() per avere un primo efficiente controllo sulla disuguaglianza tra due oggetti. Questo dovrà infatti sempre ricorrere all'invocazione di `equals()`, perché il confronto tra i valori ritornati da `hashCode()` non sarà mai utile.

Overriding di hashCode() (3)

Nel caso frequente di classi le cui istanze sono equal se e solo se coincidono nel valore (di un sottoinsieme) degli attributi, un metodo corretto, ragionevole, e semplice di fare ovverriding di `hashCode()` è quello di combinare tra loro i valori (se di tipo base) o gli hash-code (se oggetti) di tali attributi.

Tornando alla classe Punto:

```
public class Punto {
   private double x, y, z;
   public Punto(double xx, yy, zz) { x=xx; y=yy; z=zz; }
   public double getX() { return x; }
   ...
   public Object equals(Object o) {
      if (o == null) return false;
      if (!this.getClass().equals(o.getClass())) return false;
      return x=((Punto)o).x && y=((Punto)o).y && z=((Punto)o).z;
   }
   public int hashCode() { return (int)x+y+z; }
}
```

Overriding di hashCode() (4)

```
public class Punto {
  ...
  public int hashCode() { return (int)x+y+z; }
}
```

Dati due riferimenti `p1` e `p2` ad oggetti di classe Punto, `p1.hashCode()` e `p2.hashCode()` torneranno valori uguali solo se la (parte intera della) somma dei valori delle coordinate di `p1` e `p2` è la stessa. Il contratto di `hashCode()` è rispettato:

Se `p1.hashCode() != p2.hashCode()` **allora** `p1.equals(p2) == false`.

Tuttavia, il metodo tornerà lo stesso intero se invocato su due punti che, sebbene diversi, coincidano nella somma dei valori delle loro coordinate. In tali casi (il contratto di `hashCode()` non permetterebbe di inferire nulla sulla uguaglianza tra i due punti) bisogna invocare il metodo `equals()` per verificare l'uguaglianza profonda tra i due oggetti.

hashCode(): un altro esempio

Definiamo i metodi `equals()` e `hashCode()` per la classe `Segmento`. Si noti che due oggetti `s1` ed `s2` di classe `Segmento` saranno equal se uniscono la stessa coppia di punti nello spazio (ovvero oggetti di classe Punto equal tra loro) indipendentemente dall'ordinamento (ovvero se `s1.inizio.equals(s2.inizio)` e `s1.fine.equals(s2.fine)`, oppure se se `s1.inizio.equals(s2.fine)` e `s1.fine.equals(s2.inizio)`).

```
public class Segmento {
  private Punto inizio, fine;
  public Segmento(Punto i, Punto f) {  inizio = i; fine = f; }
  public Punto getInizio() { return inizio; }
  public Punto getFine() { return fine; }

  public boolean equals(Object o) {
      if (o == null) return false;
      if (!this.getClass().equals(o.getClass())) return false;
      Segmento oo = (Segmento)o;
      return (inizio.equals(oo.inizio) && fine.equals(oo.fine)) ||
                 (inizio.equals(oo.fine) && fine.equals(oo.inizio));
  }
  public int hashCode() { return inizio.hashCode()+fine.hashCode(); }
}
```

hashCode(): esempio di cliente

Supponiamo di definire una classe che realizza il tipo di dato Insieme(Punto) (fino a 10 elementi, per semplicità).

Se la classe Punto **non** fornisce l'overriding di `hashCode()`:

```
public class InsiemeDiPunti {
   private Punto elementi[];
   private int card = 0;
   public InsiemeDiPunti() {  elementi[] = new Punto[10]; }

   // SENZA USO DI HASHCODE()
   public void inserisci(Punto p) {
      if (card == 10) return;  // array pieno: non inserisco
      if (p == null) return;
      for(int i=0; i<card; i++) {
         if ( p.equals(elementi[i]) return;  // elemento gia' esistente:
                                                    // non inserisco
      }
      card++;
      elementi[card-1] = p;
   }
}
```

hashCode(): esempio di cliente (2)

Se invece la classe Punto **fornisce** l'overriding di `hashCode()`:

```
public class InsiemeDiPunti {
   ...
   // CON USO DI HASHCODE()
   public void inserisci(Punto p) {
      if (card == 10) return;  // array pieno: non inserisco
      if (p == null) return;
      for(int i=0; i<card; i++) {
         if ( p.hashCode() == elementi[i].hashCode() ) {
            if (p.equals(elementi[i]) return;  // elemento gia' esistente:
                                                                  // non inserisco
         }
      }
      card++;
      elementi[card-1] = p;
   }
}
```

hashCode() in classi derivate

Nel fare l'overriding del metodo `hashCode()` in una classe `D` derivata dalla classe `B` che fornisce l'overriding di `hashCode()`, ci si può basare sul seguente schema semplificato, nel caso (frequente) in cui l'uguaglianza profonda tra gli oggetti è basata sull'uguaglianza dei campi:

```
public int hashCode() {
  return super.hashCode() + .../* combinazione degli hash-code
         o dei valori di tipo base della classe D */
  }
}
```

Un esempio particolare

Vediamo l'overriding di `equals()` e `hashCode()` della classe `SegmentoOrientato`, derivata dalla classe `Segmento`.

Si noti che in questo caso non è possibile usare lo schema semplificato descritto poc'anzi, dato che l'uguaglianza tra due oggetti di classe SegmentoOrientato non viene determinata esclusivamente dall'uguaglianza campo a campo (cfr. `Segmento.equals()`).

In particolare, dati due oggetti `so1` e `so2` di classe `SegmentoOrientato`, questi saranno equal se uniscono la stessa coppia di punti nello stesso ordine.

Un esempio particolare (cont.)

```
public class SegmentoOrientato extends Segmento {
  private boolean daInizioAfine;
  public SegmentoOrientato(Punto in, Punto fi, boolean daInAfi) {
     super(in, fi);
     daInizioAFine = daInAfi;
  }
  public boolean equals(Object o) {
    if (!super.equals(o)) return false;
    SegmentoOrientato oo = (SegmentoOrientato)o;
    Punto inizialeThis = ((daInizioAFine)?getInizio():getFine());
    Punto inizialeO = ((oo.daInizioAFine)?oo.getInizio():oo.getFine());

    return inizialeThis.equals(inizialeO);
  }
  public int hashCode() {
    Punto inizialeThis = ((daInizioAFine)?getInizio():getFine());
    return super.hashCode() + inizialeThis.hashCode();
  }
}
```

Si ricorda che l'espressione `(condizione)?exprTrue:exprFalse` valuta `condizione` per decidere quale tra due espressioni valutare. In particolare, se `condizione == true` viene valutata `exprTrue`, altrimenti viene valutata `exprFalse`. L'intera espressione `(condizione)?exprTrue:exprFalse` ritorna il valore ritornato dall'espressione `exprTrue` o `exprFalse` valutata.

Overriding di hashCode(): riassunto

In conclusione:

- Ogni volta che si effettua overriding di equals(), va fatto anche l'overriding di hashCode();
- L'overriding di hashCode() deve essere fatto in modo che il seguente contratto sia rispettato:

  Se due oggetti hanno diverso hash-code,
  allora sono sicuramente non-equal.

- Tuttavia, l'implementazione di hashCode() deve essere anche tale da effettuare un controllo sufficientemente preciso ma efficiente, della non uguaglianza di due oggetti, ovvero deve essere tale che due oggetti diversi molto probabilmente abbiano diversi hash-code;
- Un modo semplice è quello di combinare (mediante funzioni efficientemente calcolabili, ad es. $+$, $-$, ecc.) i valori o gli hash-code degli attributi dell'oggetto di invocazione la cui uguaglianza è controllata dal metodo equals(). Questa tecnica è corretta quando l'uguaglianza profonda tra gli oggetti è data dall'uguaglianza dei valori (di un sottoinsieme) degli attributi.
- L'overriding di `hashCode()` in una classe `D` derivata dalla classe `B` che a sua volta fornisce l'overriding di `hashCode()` può usare `B.hashCode()` per calcolare l'hash-code relativo agli attributi definiti in `B`.

Esercizio 14bis

Sia data la seguente classe Java OraConFuso le cui istanze rappresentano orari relativamente ad un particolare fuso orario (rappresentato mediante un intero tra -11 e +12 – offset rispetto GMT. Per semplicità, si ignorino i fusi che differiscono da GMT di un numero non intero di ore):

```
public class OraConFuso {
  private int ore, minuti, secondi;
  private int fuso;
  public OraConFuso(int h, int m, int s, int f) {
    ore = h; minuti = m; secondi = s; fuso = f;
  }
  public int getOre() { return ore; }   // ecc.
}
```

Effettuare l'overriding delle funzioni equals() e hashCode().

Osservazione. In questo caso, l'uguaglianza profonda tra due istanze non deriva semplicemente dall'uguaglianza tra i valori degli attributi. In particolare, due istanze devono essere equal se rappresentano istanti contemporanei, sebbene espressi rispetto a fusi orari diversi. Esempio: 5:27:33 GMT+1 è equal a 3:27:33 GMT-1).

Riprogettare la classe in modo da rappresentare anche orari espressi rispetto a fusi che differiscono di un numero non intero di ore da GMT (ad es., Kathmandu in Nepal è nel fuso orario GMT+5:45 –cfr. ad es. `http://www.timeanddate.com/worldclock`.)