Uguaglianza e copia di oggetti

Sommario

1. Classi Object e Class
2. Uguaglianza superficiale e uguaglianza profonda
3. Copia superficiale e copia profonda
4. Uguaglianza e copia in classi derivate

La classe Object

Implicitamente, **tutte le classi** (predefinite o definite da programma) sono derivate, direttamente o indirettamente, dalla classe `Object`.

Di conseguenza, tutti gli oggetti, qualunque sia la classe a cui appartengono, **sono anche implicitamente istanze** della classe predefinita `Object`.

Queste sono alcune funzioni della classe `Object` di cui è utile fare overriding:

- `public boolean equals(Object)`.
- `protected Object clone()`.
- `public String toString()`.

La classe Class

- Esiste implicitamente un oggetto di classe `Class` per ogni classe `B` (di libreria o definita da utente) del programma;

- Questo oggetto può essere denotato in due modi:
  - tramite la classe `B`, con l'espressione:

    ```
    ...  B.class ... // ha tipo Class
    ```

  - tramite oggetti della classe `B`, usando la funzione `getClass()` di `Object`, ad es.:

    ```
    B b1 = new B();
    ... b1.getClass() ... // ha tipo Class
    ```

La classe Class (cont.)

- L'oggetto di classe `Class` può essere usato per verificare se due oggetti appartengono alla stessa classe, ad es.:

```
B b2 = new B();
... b1.getClass().equals(b2.getClass()) ... // vale true
```

- La classe `Class` ha una funzione dal significato particolare:

```
boolean isInstance(Object)
```

  che restituisce `true` se e solo se il suo parametro attuale è un riferimento ad oggetto di una classe *compatibile per l'assegnazione* con la stessa classe dell'oggetto di invocazione.

La funzione isInstance()

- La funzione `isInstance()` può essere usata per verificare se un oggetto è istanza di una classe.

```
... B.class.isInstance(b2) ... // vale true
```

- Al riguardo, si ricorda che un oggetto di una classe `D` derivata da una classe `B` *è oggetto anche della classe* `B`.

```
class D extends B ...
```

```
D d1 = new D();
... B.class.isInstance(d1)) ... // vale true;
```

Esercizio: cosa fa questo programma?

```
class B {}
class D extends B {}

public class Esercizio {
  public static void main(String[] args) {
    B b1 = new B();
    D d1 = new D();
    System.out.println(B.class.isInstance(d1));
    System.out.println(D.class.isInstance(b1));
  }
}
```

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: superficiale

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

Ad esempio:

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

Uguaglianza fra oggetti: 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: funzione equals()

In Java esiste un'altra maniera per verificare l'uguaglianza fra oggetti, tramite la funzione `equals()`.

`equals()` **esiste implicitamente** per ogni classe (standard, o definita dal programmatore), e *se non ridefinita*, si comporta come l'operatore '=='.

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

Uguaglianza fra oggetti: profonda

È tuttavia possibile *ridefinire* il significato della funzione `equals()`, facendo in maniera tale che verifichi l'*uguaglianza profonda* fra oggetti.

Nel seguente esempio, ciò viene fatto per la classe `B`.

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

Uguaglianza fra oggetti: profonda (cont.)

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

- `equals(Object o)` è una funzione *ereditata* dalla classe `Object`, che vogliamo ridefinire per la classe `B`.
- Tecnicamente, ciò viene effettuato mediante *overriding* di tale funzione, e comporta la definizione in `B` di una funzione con argomento (`o`) di classe `Object` (e **NON** di classe `B`) – *maggiori dettagli in seguito*.
- Il fatto che `o` sia un riferimento, di classe `Object`, ha alcune conseguenze:
  - dobbiamo essere sicuri che `o` si riferisca ad un oggetto che è stato già allocato (`o != null`);
  - dobbiamo essere sicuri che `o` si riferisca ad un oggetto della classe `B` (`getClass().equals(o.getClass())`).

Uguaglianza fra oggetti: profonda (cont.)

Alcuni commenti sulla funzione `getClass()`:

- La funzione `getClass()` è definita in `Object` e restituisce la classe dell'oggetto di invocazione (cioè, la classe più specifica di cui l'oggetto d'invocazione è istanza).

- Più precisamente, `getClass()` restituisce un oggetto della classe predefinita `Class` associato alla classe dell'oggetto di invocazione.

- Esiste un oggetto di classe `Class` per ogni classe definita nel programma.

Uguaglianza fra oggetti: profonda (cont.)

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

- Se la condizione logica dell'`if` risulta vera (ovvero l'oggetto denotato da `o` esiste ed appartiene alla classe `B`), allora possiamo:
  - definire un riferimento (non un oggetto!) `b` di classe `B`;
  - assegnare a `b` l'oggetto denotato da `o`, attraverso una conversione di tipo esplicita (`B b = (B)o`);
  - verificare l'uguaglianza tra i singoli campi della classe `B` (`return (b.x == x) && (b.y == y)`).

Uguaglianza fra oggetti: profonda (cont.)

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

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

il **cliente** di `B` deve effettuare il confronto fra oggetti usando `equals()` per l'uguaglianza profonda e '==' per quella superficiale.

```
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

L'unica eccezione a quanto detto è la classe predefinita `String`.

In `String` la funzione `equals()` è già 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: cosa fa questo programma?

```
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 Esercizio {
  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), b2 = new B(10,20);
    if (b1.equals(b2))   System.out.println("I DUE OGGETTI SONO UGUALI");
    else   System.out.println("I DUE OGGETTI SONO DIVERSI");
    stampaUguali(b1, b2);
  }
}
```

Copia di valori di un tipo base

Se vogliamo copiare un valore di un tipo base in una variabile dello stesso tipo, usiamo l'*operatore di assegnazione* '='.

Ad esempio:

```
int a = 4, b;
b = a;
```

Copia fra oggetti: superficiale

Se usiamo '=' per copiare **due oggetti**, stiamo effettuando la copia *superficiale*.

Ad esempio:

```
class C {
  int x, y;
}
void F() {
    C c1 = new C(), c2;                              // 1
    c1.x = 4; c1.y = 5;                              // 2
    System.out.println("c1.x: " + c1.x + ", c1.y: " + c1.y);
    c2 = c1;    // COPIA SUPERFICIALE                // 3
    System.out.println("c2.x: " + c2.x + ", c2.y: " + c2.y);
    c2.x = -70; // SIDE-EFFECT                       // 4
    System.out.println("c1.x: " + c1.x + ", c1.y: " + c1.y);
}
```

Copia fra oggetti: superficiale (cont.)

L'operatore '=' effettua una copia fra *i valori dei riferimenti*, ovvero fra i due indirizzi di memoria in cui si trovano gli oggetti.

Riassumendo, diciamo che:

1. '=' effettua la copia *superficiale*,

2. in quanto tale **non crea un nuovo oggetto**,

3. a seguito dell'assegnazione, i due riferimenti `c1` e `c2` *sono uguali superficialmente*,

4. ogni azione sul riferimento `c2` si ripercuote sull'oggetto a cui si riferisce anche `c1`.

Copia fra oggetti: clone() e copia profonda

In Java esiste un'altra maniera per copiare oggetti, la funzione `clone()`.

`clone()` **non esiste implicitamente** per una classe (*maggiori dettagli in seguito*). Se lo desideriamo, possiamo **ridefinirla**, facendo in maniera tale che effettui la *copia profonda* fra oggetti. Nel seguente esempio, ciò viene fatto per la classe `B`.

```
class B implements Cloneable {
  int x, y;
  public Object clone() {
    try {
      B t = (B)super.clone(); // Object.clone copia campo a campo
      return t;
    } catch (CloneNotSupportedException e) {
      // non puo' accadere, ma va comunque gestito
      throw new InternalError(e.toString());
    }
  }
}
```

Copia fra oggetti: copia profonda (cont.)

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

- `clone()` è una funzione *ereditata* dalla classe `Object` (ma non direttamente disponibile per i clienti: *maggiori dettagli in seguito*) che vogliamo ridefinire per la classe `B`.

- Tecnicamente, ciò viene effettuato mediante *overriding* di tale funzione, e comporta la definizione in `B` di una funzione che restituisce un riferimento di classe `Object` (e **NON** di classe `B`) – *maggiori dettagli in seguito*.

- Dobbiamo inoltre dichiarare che `B` *implementa l'interfaccia* `Cloneable` – *maggiori dettagli in seguito*.

Copia fra oggetti: copia profonda (cont.)

Altri commenti sulla funzione `clone()` ridefinita per la classe `B`:

- Mediante `super.clone()` viene invocata la funzione `clone()` di `Object`. Questa funzione crea (allocandolo dinamicamente) l'*oggetto* ed esegue una **copia superficiale dei campi** (cioè mediante '=') dell'oggetto di invocazione, indipendentemente dalla classe a cui questo appartiene.

- Il riferimento restituito da `super.clone()`, che è di tipo `Object`, viene convertito, mediante *casting* in un riferimento di tipo `B` (`(B)super.clone()`).

- Infine, dobbiamo trattare in modo opportuno l'eccezione (*checked exception*) `CloneNotSupportedException` che `clone()` di `Object` genera se invocata su un oggetto di una classe che non implementa l'interfaccia `Cloneable`.

Copia fra oggetti: copia profonda (cont.)

Riassumendo, se desideriamo che per una classe `B` si possa effettuare la copia profonda fra oggetti, allora:

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

il **cliente** di `B` deve effettuare la copia fra oggetti usando `clone()` per la copia profonda e '=' per quella superficiale.

```
B b1 = new B();
b1.x = 10; b1.y = 20;
B b2 = (B)b1.clone();
System.out.println("b2.x: " + b2.x + ", b2.y: " + b2.y);
```

Copia profonda: classe String

Come per le altre classi predefinite, la funzione `clone` non esiste per la classe `String`.

Se vogliamo fare una copia profonda di un oggetto di tale classe, possiamo utilizzare, mediante `new`, un suo costruttore speciale, che accetta un argomento di tipo `String`.

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

s2 = new String(s1); // uso del costruttore con argomento String
                     // ora s2 si riferisce ad una copia profonda di s1
```

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 d.z == z;
    }
    else return false;
  }
}
```

Uguaglianza profonda in classi derivate (cont

- `D.equals()` delega a `super.equals()` (cioè `B.equals()`) alcuni controlli (**riuso**):
  - che l'oggetto a cui si riferisce il parametro attuale sia stato allocato;
  - 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: cosa fa questo programma?

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

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

Copia profonda in classi derivate

Quando una classe `B` ha dichiarato pubblica `clone()`, tutte le classi da essa derivate (direttamente o indirettamente) **devono** supportare la clonazione (non è più possibile "nascondere" `clone()`).

Per supportarla correttamente le classi derivate devono fare overriding di `clone()` secondo lo schema seguente.

```
public class D extends B {
  // ...
  public Object clone() {
    D d = (D)super.clone();
      // codice eventuale per campi di D che richiedono copie speciali
    return d;
  }
  // ...
}
```

Copia profonda in classi derivate (cont.)

- Una classe derivata da una classe che implementa l'interfaccia `Cloneable` (o qualsiasi altra interfaccia), implementa anch'essa tale interfaccia.

- La chiamata a `super.clone()` è **indispensabile**.

  Essa invoca la funzione `clone()` della classe base, la quale a sua volta chiama `super.clone()`, e così via fino ad arrivare a `clone()` della classe `Object` che è l'unica funzione in grado di creare (allocandolo dinamicamente) l'oggetto clone.

  Tutte le altre invocazioni di `clone()` lungo la catena di ereditarietà si occupano in modo opportuno di operare sui campi a cui hanno accesso.

  Si noti che per copiare correttamente gli eventuali campi privati è indispensabile operare sugli stessi attraverso la classe che li definisce.

Copia profonda in classi derivate: esempio

```
class B implements Cloneable {
  protected int x, y;
  public Object clone() { // ...
  // ...
}
```

```
class C implements Cloneable {
  private int w;
  public Object clone() { // ...
  // ...
}
```

```
class D extends B {
  protected int z;            // TIPO BASE
  protected C c;              // RIFERIMENTO A OGGETTO
  public Object clone() {
    D d = (D)super.clone();   // COPIA SUPERFICIALE: OK PER z, NON PER c
    d.c = (C)c.clone();       // NECESSARIO PER COPIA PROFONDA DI c
    return d;
  }
  // ...
}
```