

Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 9: Uguaglianza e copia degli oggetti

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Ridefinire `equals` · Il metodo `hashCode`
Copia superficiale e profonda · Ridefinire `clone`
Oggetti mutabili e immutabili · Quando ridefinirli

Indice

1	Ridefinire equals	1
2	Il metodo hashCode	3
3	equals e hashCode nelle derivate	4
4	Copia superficiale e profonda	5
5	Ridefinire clone	6
6	clone nelle derivate	7
7	Oggetti mutabili e immutabili	8
8	Quando ridefinire equals , hashCode e clone	9
9	Esercizi	10
10	Soluzioni	13

1 Ridefinire `equals`

Per l'uguaglianza fra oggetti si sceglie il significato (Capitolo 4, Sezione 7):

- **superficiale**: due riferimenti denotano lo stesso oggetto;
- **profonda**: le informazioni rilevanti dei due oggetti sono uguali.

Due banconote da 10 euro sono uguali profondamente, perché hanno lo stesso valore, ma non superficialmente, perché sono due pezzi di carta distinti.

```
public class C {  
    private int x, y;  
    public C(int m, int n) { x = m; y = n; }  
}  
...  
C c1 = new C(4, 5);  
C c2 = new C(4, 5);  
System.out.println(c1.equals(c2));    // false
```

`c1` e `c2` sono uguali profondamente ma non superficialmente. `C` non ridefinisce `equals`, e quello ereditato da `Object` si comporta come `==`: verifica l'uguaglianza superficiale.

Proposizione – Ridefinire `equals`

Per l'uguaglianza profonda si ridefinisce `equals` sempre con lo stesso schema:

1. la segnatura è `public boolean equals(Object o)`, con `@Override`;
2. se `o` è `null`, gli oggetti sono diversi;
3. se `o` non è della stessa classe, gli oggetti sono diversi;
4. il cast di `o` alla classe;
5. il confronto dei campi rilevanti.

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

Il parametro è di tipo `Object`: solo così è un overriding di `equals` di `Object`. Dopo il controllo con `getClass` (Capitolo 8, Sezione 5) il cast è sicuro, e `b` dà accesso ai campi dell'altro oggetto. Nel confronto:

- i campi di tipo primitivo si confrontano con `==`;
- i campi che sono oggetti si confrontano a loro volta con `equals`.

Osservazione – `getClass` e non `instanceof`

`equals` deve essere **simmetrico**: se `a.equals(b)` è vero, lo è anche `b.equals(a)`. Con `instanceof` un oggetto della base può risultare uguale a uno della derivata ma non viceversa, mentre con `getClass` oggetti di classi diverse non sono mai uguali (Esercizio 1).

Nota – Overloading invece di overriding

Un `equals(B b)` con il parametro di tipo `B` non ridefinisce `equals(Object)` ma lo sovraccarica, e la classe ha due metodi `equals`. Con `@Override` il compilatore segnala l'errore: *method does not override or implement a method from a supertype*.

```

public 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 EqualsOverloaded {
    static void stampaUguali(Object o1, Object o2) {
        if (o1.equals(o2)) System.out.println("uguali");
        else System.out.println("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("uguali"); // uguali
        else System.out.println("diversi");
        stampaUguali(b1, b2); // diversi
    }
}

```

Stessi oggetti e stesso confronto danno due risposte, perché il metodo si sceglie in due tempi (Capitolo 7, Sezione 6):

1. durante la compilazione si sceglie la segnatura dai tipi statici degli argomenti. In `b1.equals(b2)` l'argomento è di tipo `B`, e si sceglie `equals(B)`. In `o1.equals(o2)` è di tipo `Object`, e si sceglie `equals(Object)`.
2. durante l'esecuzione il late binding cerca la versione di quella segnatura nella classe dell'oggetto. `B` non ridefinisce `equals(Object)`, quindi si esegue quella di `Object`, che confronta i riferimenti.

2 Il metodo hashCode

Proposizione – Vincolo fra equals e hashCode

Chi ridefinisce `equals` deve ridefinire anche `public int hashCode()`, in modo che oggetti uguali abbiano lo stesso codice:

$$o1.equals(o2) \Rightarrow o1.hashCode() == o2.hashCode()$$

Il codice serve a cercare in fretta. È come lo scaffale di una biblioteca: per un autore che comincia per R si va allo scaffale R, e lì si cerca il libro con **equals**. Se due oggetti uguali avessero codici diversi, si cercherebbe sullo scaffale sbagliato.

Il codice si calcola dai campi che **equals** confronta, così oggetti uguali danno lo stesso intero. Per **B** si aggiunge:

```
@Override
public int hashCode() {
    return x + 37 * y;    // dai campi x e y, quelli di equals
}
```

Il punto (1, 2) ha codice $1 + 37 \cdot 2 = 75$, il punto (2, 1) ha $2 + 37 = 39$. Con $x + y$ avrebbero lo stesso codice 3, come tutti i punti con la stessa somma. Il vincolo resta rispettato, ma molti oggetti finiscono sullo stesso scaffale, e la ricerca rallenta. Moltiplicare per un numero primo come 37 distribuisce meglio i codici.

Nota – **Objects.hash**

Objects.hash(x, y), del pacchetto **java.util**, combina i campi in un codice senza doverci pensare: **return Objects.hash(x, y);**.

3 **equals** e **hashCode** nelle derivate

Una derivata di una classe che ridefinisce **equals** confronta prima i campi della base, con **super.equals(o)**, e poi i propri. È lo stesso riuso di **super.toString()** (Capitolo 8, Sezione 4), e qui è anche obbligato: i campi della base sono spesso **private**, e la derivata non li può leggere.

```
public class D extends B {
    private int z;
    public D(int a, int b, int c) { super(a, b); z = c; }
    @Override
    public boolean equals(Object o) {
        if (super.equals(o)) { // o non è null, è un D, ha gli stessi x e y
            D d = (D) o;
            return z == d.z; // i campi propri
        }
        else return false;
    }
    @Override
    public int hashCode() {
        return super.hashCode() + 53 * z;
    }
}
```

Se `super.equals(o)` è vero, `o` non è `null`, ha la stessa classe dell'oggetto, cioè `D`, e ha gli stessi `x` e `y`: il cast a `D` è sicuro. `equals` di `D` usa `x`, `y` e `z`, e così anche il suo `hashCode`, che riusa quello di `B`.

4 Copia superficiale e profonda

Definizione – Copia superficiale e copia profonda

- La **copia superficiale** copia il riferimento: dopo `c2 = c1` ci sono due variabili e un solo oggetto.
- La **copia profonda** crea un oggetto nuovo, un **clone**, con le stesse informazioni dell'originale ma un riferimento diverso.

Con la copia superficiale una modifica fatta attraverso una variabile si vede dall'altra: è il side effect del Capitolo 4, Sezione 8. Qui `C` ha un campo `x`, con `getX` e `setX`:

```
C c1 = new C(10);  
C c2 = c1;          // copia superficiale  
c2.setX(100);  
System.out.println(c1.getX()); // 100
```

Il clone invece è indipendente: le modifiche alla copia non toccano l'originale, e viceversa.

5 Ridefinire `clone`

Per la copia profonda `Object` ha il metodo `clone()`, che è **protected**: è ereditato da tutti gli oggetti, ma non è accessibile ai clienti, e il compilatore segnala *clone() has protected access in Object*. Una classe che vuole offrire la copia lo ridefinisce **public**.

Non sempre una copia ha senso: in un sistema con persone e codici fiscali, il clone di una persona sarebbe un duplicato della stessa persona (Sezione 8). Serve invece per provare una modifica senza toccare l'originale, come vedere come cambia la media dei voti aggiungendone uno (Esercizio 2).

Proposizione – Ridefinire `clone`

1. La classe dichiara **implements Cloneable**, un'interfaccia senza metodi che **marca** gli oggetti della classe come clonabili.
2. `clone` si ridefinisce **public**: l'overriding può allargare l'accesso (Capitolo 7, Sezione 5).
3. `super.clone()`, cioè il `clone` di `Object`, crea un oggetto nuovo della stessa classe dell'oggetto di invocazione e ne copia i campi uno per uno, in modo superficiale.
4. Il cast porta il risultato, di tipo `Object`, al tipo della classe.
5. I campi che sono array o altri oggetti modificabili si copiano in profondità.

```
public class B implements Cloneable {
    private int x, y;
    ...
    @Override
    public Object clone() {
        try {
            B b = (B) super.clone();
            // copia profonda dei campi che la richiedono: array, liste, ...
            return b;
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e);
        }
    }
}
```

La copia di `Object` è la copia esatta della memoria occupata dall'oggetto, e non si ottiene in nessun altro modo: per questo ogni `clone` deve arrivare, attraverso `super.clone()`, a quello di `Object`. Ma un campo che è un riferimento viene copiato come riferimento, e così originale e clone condividerebbero lo stesso array. Il passo 5 rimedia.

L'eccezione. Se la classe non implementa `Cloneable`, `super.clone()` lancia l'eccezione `CloneNotSupportedException`, cioè segnala l'errore invece di restituire un risultato. Java obbliga a gestirla con `try` e `catch`, e senza il compilatore segnala *unreported exception*. Qui la classe implementa `Cloneable`, quindi l'eccezione non può accadere, e il `catch` la trasforma in un `InternalError`.

Il cliente chiede la copia esplicitamente, con il cast: `B b2 = (B) b1.clone();`.

6 clone nelle derivate

La derivata chiama il `clone` della base, che sa quali dei suoi campi, spesso `private`, vanno copiati in profondità, e poi copia i propri.

```
public class D extends B {
    private int z;
    ...
    @Override
    public Object clone() {
        D d = (D) super.clone(); // il clone di B
        // copia profonda dei campi propri che la richiedono
        return d;
    }
}
```

Osservazione – Perché la derivata è più semplice

- **D** non scrive `implements Cloneable`, ma è clonabile: una derivata eredita anche le interfacce implementate dalla base.
- Il cast a **D** è sicuro: `clone` di **B** arriva a quello di **Object**, che crea un oggetto della classe dell'oggetto di invocazione, un **D**.
- Non serve `try`: `clone` di **B** non lancia l'eccezione, perché la gestisce già.

7 Oggetti mutabili e immutabili

Definizione – Oggetti mutabili e immutabili

Lo **stato** di un oggetto sono i valori dei suoi campi in un dato momento.

- Un oggetto è **mutabile** se il suo stato può cambiare durante l'esecuzione, cioè se la classe ha metodi che fanno **side effect**, modificandone i campi.
- Un oggetto è **immutabile** se il suo stato non può cambiare, cioè nessun metodo fa side effect.

Immutabili. I metodi leggono i campi, calcolano e restituiscono risultati, ma non toccano lo stato. Se serve un oggetto modificato, come un punto spostato, se ne crea uno nuovo, e l'originale resta com'è. Così `s.toUpperCase()`, da solo, lascia `s` uguale, e per avere la stringa in maiuscolo si scrive `s = s.toUpperCase();`. Gli oggetti immutabili rappresentano **valori**, come una data, un importo o un numero complesso.

Mutabili. I clienti possono cambiarne lo stato con i metodi, per esempio con un `set`: è il caso più comune. Rappresentano **entità**, come una persona, uno studente, un conto corrente, che cambiano nel tempo restando se stesse: chi cambia residenza è sempre la stessa persona. Per efficienza si usano mutabili anche valori complessi, come le collezioni.

8 Quando ridefinire `equals`, `hashCode` e `clone`

Proposizione – Che cosa ridefinire

La scelta dipende da due domande: conta l'identità dell'oggetto o il suo contenuto? Può cambiare nel tempo?

Oggetti	Conta	<code>equals</code> , <code>hashCode</code>	<code>clone</code>
entità mutabili	l'identità	di <code>Object</code>	di <code>Object</code> , inaccessibile
valori mutabili, come le collezioni	il contenuto	ridefiniti	ridefinito, con copia profonda, se si vuole permettere la copia
valori immutabili	il contenuto	ridefiniti	di <code>Object</code> : basta condividere il riferimento

Entità. Il riferimento è l'identità, ed è ciò che conta: due persone di nome Mario Rossi, nate lo stesso giorno nella stessa città, sono due persone. `equals` di `Object` è già giusto. Ridefinirlo sui dati sarebbe un errore: due studenti omonimi nati lo stesso giorno risulterebbero uguali, e un insieme di iscritti a un esame, che non ha duplicati, scarterebbe il secondo. Anche la copia di un'entità non ha senso, perché un conto corrente copiato sarebbe un secondo conto con lo stesso numero e lo stesso saldo.

Valori. Due insiemi con gli stessi numeri sono lo stesso insieme, chiunque li abbia creati e in qualunque ordine siano stati inseriti gli elementi: si ridefinisce `equals`, e con esso `hashCode`. Se un valore mutabile si può copiare, la copia deve essere profonda, altrimenti originale e copia condividerebbero la struttura interna. Un valore immutabile non si copia mai: tanti riferimenti possono denotare lo stesso oggetto, perché nessuno lo può modificare.

Riepilogo

- **equals**: `public boolean equals(Object o)` con `@Override`: `null`, `getClass`, `cast`, campi.
- **Overloading di equals**: con il parametro di un'altra classe non ridefinisce `equals(Object)`, e il risultato dipende dal tipo statico.
- **hashCode**: oggetti uguali hanno lo stesso codice, calcolato dai campi di `equals`.
- **Derivate**: `super.equals(o)`, `super.hashCode()`, `super.clone()`, poi i campi propri.
- **Copia**: superficiale copia il riferimento, profonda crea un clone indipendente.
- **clone**: implements `Cloneable`, `public`, `super.clone()`, `cast`, copia profonda dei campi modificabili.
- **Mutabili e immutabili**: i metodi cambiano lo stato, oppure creano oggetti nuovi.
- **Quando ridefinire**: per le entità niente, per i valori `equals` e `hashCode`, `clone` solo per i valori mutabili.

9 Esercizi

Esercizio 1 – La simmetria di equals

Queste classi controllano la classe con `instanceof` invece che con `getClass`. Che cosa stampano `b.equals(d)` e `d.equals(b)`, con `B b = new B(1);` e `D d = new D(1, 2);`? Quale regola di `equals` non è rispettata?

```
public class B {
    private int x;
    public B(int a) { x = a; }
    @Override
    public boolean equals(Object o) {
        if (o instanceof B) return x == ((B) o).x;
        else return false;
    }
}

public class D extends B {
    private int z;
    public D(int a, int c) { super(a); z = c; }
    @Override
    public boolean equals(Object o) {
        if (o instanceof D) return super.equals(o) && z == ((D) o).z;
        else return false;
    }
}
```

```
}
}
```

Soluzione a pagina 13 →

Esercizio 2 – La copia dei voti

La classe `Voti` ridefinisce `clone` senza copiare l'array.

```
public class Voti implements Cloneable {
    private int[] voti;
    private int n; // quanti voti sono stati inseriti
    public Voti(int max) { voti = new int[max]; }
    public void aggiungi(int v) { voti[n] = v; n++; }
    public void correggi(int i, int v) { voti[i] = v; }
    public double media() {
        int somma = 0;
        for (int i = 0; i < n; i++) somma = somma + voti[i];
        return (double) somma / n;
    }
    @Override
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e);
        }
    }
}
```

Che cosa stampa il codice seguente? Correggi `clone` perché stampi 26,0.

```
Voti v = new Voti(10);
v.aggiungi(24);
v.aggiungi(28);
Voti w = (Voti) v.clone();
w.correggi(0, 18);
System.out.println(v.media());
```

Soluzione a pagina 13 →

Esercizio 3 – Entità e valori

Per ciascuna classe, quali fra `equals`, `hashCode` e `clone` vanno ridefiniti, e perché?

1. `ContoCorrente`, con numero e saldo, e i metodi `deposita` e `preleva`.
2. `Data`, con giorno, mese e anno fissati dal costruttore e i soli getter.
3. `InsiemeDiInteri`, con i metodi `aggiungi` e `togli`.

[Soluzione a pagina 13 →](#)

10 Soluzioni

Soluzione dell'Esercizio 1 – La simmetria di equals

b.equals(d) stampa true, **d.equals(b)** stampa false:

1. in **b.equals(d)** si esegue **equals** di **B**. Un **D** è un **B**, quindi **d instanceof B** è vero, e le due **x** valgono 1.
2. in **d.equals(b)** si esegue **equals** di **D**, e **b instanceof D** è falso, perché un **B** non è un **D**.

Non è rispettata la simmetria. Con **getClass().equals(o.getClass())** entrambe le chiamate danno **false**, perché le classi sono diverse.

← Torna all'Esercizio 1, pagina 10

Soluzione dell'Esercizio 2 – La copia dei voti

Stampa **23.0**. **super.clone()** copia i campi in modo superficiale, quindi **w.voti** è lo stesso array di **v.voti**. **w.correggi(0, 18)** cambia il primo voto anche per **v**, che ha media $(18 + 28)/2 = 23$ invece di $(24 + 28)/2 = 26$.

L'array va copiato in profondità, con **Arrays.copyOf** (Capitolo 5, Sezione 6), che crea un array nuovo con gli stessi elementi:

```
@Override
public Object clone() {
    try {
        Voti c = (Voti) super.clone();
        c.voti = Arrays.copyOf(voti, voti.length);    // copia profonda
        return c;
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }
}
```

Con **import java.util.Arrays;** il programma stampa **26.0**, e **w.media()** vale 23. Il campo **n** è un **int**, e la copia superficiale basta.

← Torna all'Esercizio 2, pagina 11

Soluzione dell'Esercizio 3 – Entità e valori

1. Un conto corrente è un'entità mutabile, e conta l'identità: nessuno dei tre. Due conti con lo stesso saldo sono due conti, e **equals** di **Object** li distingue.

Un clone sarebbe un secondo conto con lo stesso numero, quindi **clone** resta quello protetto di **Object**.

2. Una **data** è un valore immutabile, e conta il contenuto: **equals** e **hashCode**, dai tre campi. **clone** no, perché nessuno può modificare la **data** e basta condividere il riferimento.
3. Un **insieme** è un valore mutabile: **equals** e **hashCode**, perché due insiemi con gli stessi elementi sono uguali, e **clone** con la copia profonda della struttura che contiene gli elementi, se si vuole permettere la copia.

[← Torna all'Esercizio 3, pagina 12](#)