

Concetto di uguaglianza

Essendo Java un linguaggio orientato ad oggetti, può risultare non scontato il concetto di uguaglianza tra due oggetti o tra due attributi di essi. Si analizzano i seguenti casi:

Uguaglianza di due tipi primitivi

Nel caso dell'uguaglianza di due tipi primitivi, **non sorgono particolari problemi**, quindi il risultato è piuttosto scontato:

```
public class Test {  
    public static void main(String[] a){ new Test(); }  
    Test() {  
        int k1 = 1;  
        int k2 = 1;  
        System.out.println(k1==k2) ;  
    }  
}
```

true

Verificando quindi l'uguaglianza tra due tipi primitivi, se il valore assunto da essi è lo stesso seguirà che anche le due variabili saranno le stesse.

Assegnazione di due primitivi

Si consideri il caso in cui venga esplicitato il valore di un tipo primitivo e ad il secondo venga attribuito lo stesso valore del primo dopo un'operazione di assegnazione. Segue che Java ritornerà come output **un'uguaglianza verificata**. Si veda l'esempio sotto:

```

public class Test {
    public static void main(String[] a){new Test();}
    Test() {
        int k1 = 1;
        int k2 = k1;
        System.out.println(k1==k2);
    }
}

```

true

Assegnazione di due oggetti

Si prenda l'esempio direttamente descritto sopra ma lo si adatti ad un'assegnazione tra due oggetti con i propri attributi. Segue che anche in questo caso **l'uguaglianza è verificata**.

```

public class Test {
    public static void main(String[] a){new Test();}
    Test() {
        P p1=new P();
        p1.x=1; p1.y=2;
        P p2=p1;
        System.out.println(p1==p2);
    }
}

```

true

Uguaglianza tra oggetti

Si osserva che nel caso in cui vengano creati due oggetti della stessa classe e vengano assegnati gli stessi valori ai loro attributi, **NON ne consegue la diretta uguaglianza tra i due oggetti**.

```
public class Test {  
    public static void main(String[] a){new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p1==p2);  
    }  
}
```

false

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y);  
    }  
}
```

Per quale motivo in questo caso non vale mentre nei 3 esempi sopra l'uguaglianza è rispettata? Troviamo una risposta nel metodo di **allocazione della memoria** utilizzato dal linguaggio Java. Di base l'allocazione di un nuovo oggetto nella memoria del programma **avviene per indirizzo**, quindi al momento della comparazione di due oggetti viene in realtà osservato l'indirizzo in memoria di questi due.

Quando, come nell'esempio 3, viene assegnato a p2 l'oggetto p1, tramite l'operatore "=" quello che viene in realtà fatto è **associare a p2 l'indirizzo di p1**. Una dimostrazione di questo fatto è il seguente esempio: supponendo di variare il valore di un attributo di p1, se si volesse stampare p2 assegnato a p1 prima della modifica di quest'ultimo, la modifica comunque si riverbererebbe su p2 perchè appunto fa riferimento all'**indirizzo di p1**.

Quindi come si può verificare l'uguaglianza? Operare alla classica maniera con la clausola: “ **p1 == p2** ” è **formalmente sbagliato** perché così in realtà si sta **verificando se p1 e p2 puntano allo stesso oggetto**, non se hanno lo stesso valore.

A questo punto potrebbe venire spontaneo pensare che sia corretto utilizzare il metodo **equals()** legato all'oggetto e predefinito, ma in realtà come esplicitato nella Guideline di Java, il metodo **equals()** ritorna **true** solo se due oggetti si riferiscono allo stesso oggetto (hanno stesso indirizzo), segue che siamo punto a capo con il nostro problema.

Proprio per questo motivo si dice che il metodo **equals()** vada ridefinito, quindi venga fatto un **Override** del metodo secondo determinati criteri scelti dal programmatore e dalle sue necessità. Ricordiamo che il metodo **equals** nella classe **Object** è definito come: **boolean equals(Object var)** quindi nella sua ridefinizione va usata la stessa identica firma.

Ridefinizione del metodo “equals”

Un primo passo per stabilire come ridefinire il metodo **equals** è fondamentalmente comprendere che azione esso compie: svolge un confronto tra due Oggetti.

Secondariamente si devono valutare i casi limite ossia:

- cosa succede se i due oggetti appartengono a classi diverse?
- cosa succede se uno dei due oggetti è inizializzato a **null**?
- cosa succede se uno dei due oggetti appartiene ad una sottoclasse?

Problema 1) Oggetti di classi diverse

In relazione al fatto che si stanno confrontando due attributi di due oggetti è inevitabile che se è verificato che l'oggetto preso come parametro dal metodo è del tipo della classe chiamante il metodo, sarà possibile effettuare un cast. Quindi una prima discriminante che ritorna una prima sentenza è l'utilizzo di **instanceof** per determinare se un oggetto passato è almeno un'istanza di quello che si sta controllando.

Problema 2) Oggetti inizializzati a null

Semplicemente per ovviare a questa condizione che renderebbe di fatto impossibile la mia comparazione, si posiziona un “ **if** ” di controllo all’inizio così da uscire direttamente dal metodo ritornando **false** se si ha a che fare con un oggetto null.

Problema 3) Presenza di istanze di sottoclassi

Per risolvere anche questa casistica che si osserva essere un’evoluzione del problema 2), è necessario sostituire l’**instanceof** con un controllore che mi ritorna la classe dell’oggetto passato come parametro. NOTA: anche l’utilizzo di **instanceof** potrebbe essere corretto ma questo dipende dalle richieste dell’applicazione.

La versione finale si presenta quindi come:

E le sottoclassi?

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y);  
    }  
    public boolean equals(Object var){  
        if(var==null) return false;  
        if (var.getClass() != this.getClass())  
            return false;  
        return (x==((P)var).x && y==((P)var).y)  
    }  
}
```

`getClass()` è definito su `Object` e dunque presente (ereditato) in ogni classe

Proprietà del metodo equals

Nonostante il metodo **equals** possa essere ridefinito a proprio piacere per adattarsi meglio all'applicazione che si sta sviluppando, è fondamentale che rispetti alcune caratteristiche chiave che rendono sensato il suo utilizzo, equals deve godere della proprietà:

- 1) **riflessiva**: per ogni riferimento non nullo: x, x.equals(x) deve ritornare **true**.
- 2) **simmetrica**: per ogni riferimento x, y, non nulli, x.equals(y) ritorna **true** se e solo se y.equals(x) ritorna **true**.
- 3) **transitiva**: per ogni riferimento x, y, z, non nulli, se x.equals(y) e y.equals(z) ritornano **true** allora anche x.equals(z) ritorna **true**.
- 4) **consistenza**: per ogni riferimento non nullo x, y, invocazioni x.equals(y) ritornano lo stesso valore fino al primo cambio nell'algoritmo.
- 5) per ogni riferimento non nullo x, x.equals(null) ritorna **false**.

Il metodo Hashcode()

La classe Object mette a disposizione anche un metodo **hashCode()** strettamente legato al metodo equals. Hashcode() è sostanzialmente una mappatura che mappa **un oggetto su un intero**, quindi ad ogni oggetto corrisponde un suo **hash** identificativo. L'unico problema è che possono crearsi **collisioni** che fanno sì che ad oggetti diversi venga assegnato lo stesso hash.

Anche Hashcode() deve rispettare determinate proprietà come:

- Se invocato più di una volta sullo stesso oggetto, deve ritornare sempre lo stesso intero. Questo vale a livello di invocazione singola perché nel corso del programma gli oggetti possono essere creati e distrutti, variando il loro hashcode.
- Se due oggetti sono uguali secondo il metodo equals, allora l'hashcode ritornato deve essere uguale.
- Non è richiesto che a due oggetti diversi (secondo equals) corrispondano hashcode differenti.

Segue inevitabilmente che **una classe che ridefinisce il metodo equals**, deve per forza **ridefinire anche il metodo hashCode()**.

Una semplice regola da ricordare è:

- a) Oggetti uguali → hashCode uguali
- b) hashCode diversi → Oggetti diversi

Quindi ricapitolando è fondamentale **sfruttare le proprietà di hashCode()** integrandole nel metodo equals, per la ricerca di oggetti uguali. E' importante anche implementare il proprio hashCode ricordando che non è compito facile.

```
import java.util.*;

public class Test {
    Set<Element> s = new HashSet<>();
    public static void main(String[] args) { new Test(); }
    public Test() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            s.add(new Element(r.nextInt(10)));
        System.out.println("Set elements: " + s);
        System.out.print("Hashcodes: ");
        for (Element t : s)
            System.out.print(t.hashCode() + " ");
        System.out.println("");
        Element toSearch = new Element(5);
        System.out.print("toSearch in set? ");
        if (s.contains(toSearch)) System.out.println("yes");
        else System.out.println("no");
        System.out.println("HashCode of toSearch: " + toSearch.hashCode());
    }
}
```

public int hashCode() { return Objects.hash(x); }

Set elements: [1, 2, 5, 7, 8, 0]

Hashcodes:
32 33 36 38 39 31

toSearch in set? yes

HashCode of toSearch: 36

NOTA: il metodo contains attiva implicitamente il metodo equals che è stato ridefinito in precedenza come descritto all'inizio.

Ridefinendo l'hashCode come suggerito nel box superiore, l'Element "toSearch" presenterà un hashCode() uguale a quello di uno degli elementi nel set.

Insiemi ordinati e comparable

L'interfaccia **Comparable<T>** consente di definire un **ordinamento totale** (o naturale) fra oggetti che la implementano. Tale interfaccia **definisce un unico metodo: int compareTo (T o)** che ritorna:

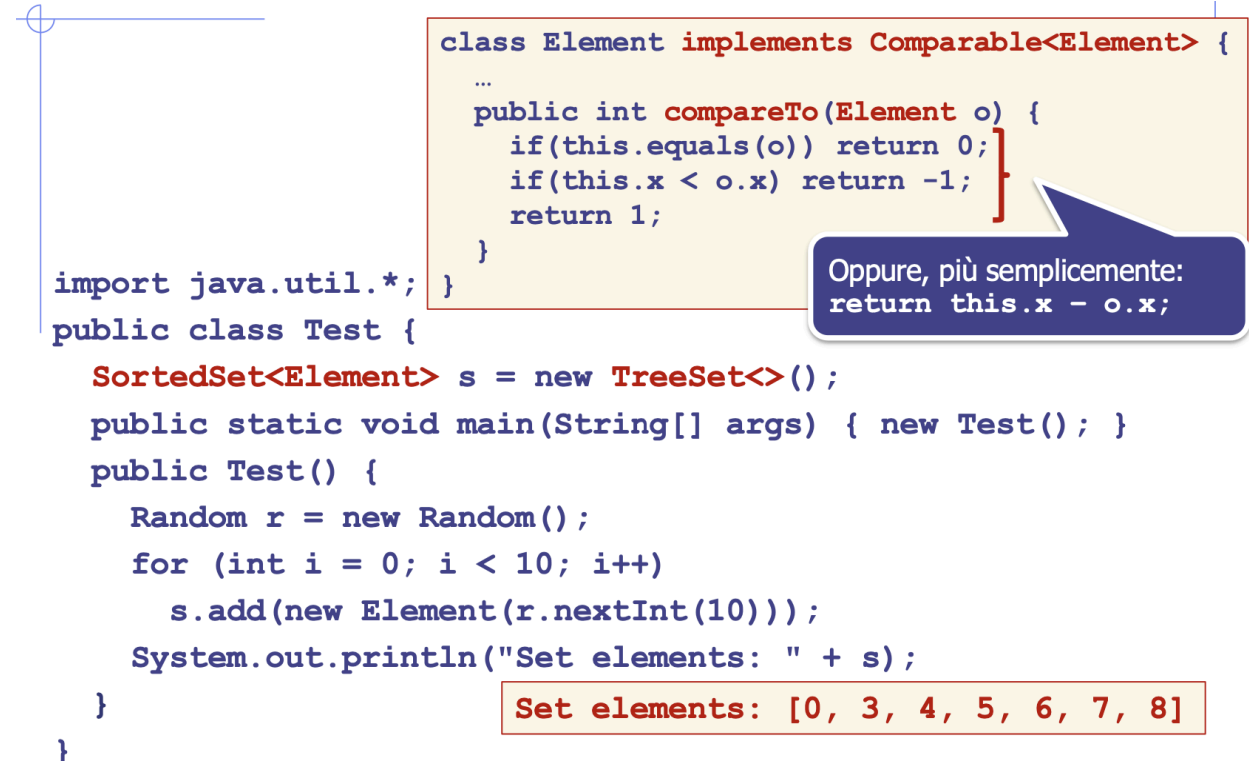
- un intero negativo se this è minore di o

- un intero positivo se this è maggiore di o
- 0 se this è uguale a o

Anche comparable deve godere di alcune proprietà fondamentali che lo rendono efficace e sicuro:

- 1) per ogni x e y deve valere: $-\text{sgn}(x.\text{compareTo}(y)) == \text{sgn}(y.\text{compareTo}(x))$
- 2) la relazione deve essere **transitiva** ovvero: $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0) \rightarrow x.\text{compareTo}(z) > 0$.
- 3) per ogni x,y,z deve valere: se $x.\text{compareTo}(y) == 0 \rightarrow \text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$
- 4) generalmente è fortemente consigliato, anche se non strettamente richiesto che: $(x.\text{compareTo}(y) == 0) == x.\text{equals}(y)$

Un primo esempio dell'**utilizzo di comparable** è il seguente:



```

import java.util.*;

public class Test {
    SortedSet<Element> s = new TreeSet<>();
    public static void main(String[] args) { new Test(); }
    public Test() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            s.add(new Element(r.nextInt(10)));
        System.out.println("Set elements: " + s);
    }
}

```

```

class Element implements Comparable<Element> {
    ...
    public int compareTo(Element o) {
        if(this.equals(o)) return 0;
        if(this.x < o.x) return -1;
        return 1;
    }
}

```

Oppure, più semplicemente:
return this.x - o.x;

```

Set elements: [0, 3, 4, 5, 6, 7, 8]

```

Notiamo che utilizzando un **sortedSet** di **Element** stiamo già implicitamente chiamando il **comparable** proprio di Element e Java si occupa di chiamarlo nel momento dell'inserimento. Quindi, il Comparable che agisce sul set, è quello che abbiamo ridefinito noi.

Se cercassimo di inserire in un **sortedSet** o **sortedMap** un elemento che non implementa un Comparable allora otterremmo un'eccezione **RUNTIME ERROR**, perché chiaramente il compilatore in compilazione non può sapere se il metodo nella classe dell'oggetto inserito è ridefinito o meno.

Se un oggetto qualsiasi implementa il metodo Comparable e tale oggetto popoli una List normale o un Array, allora quando verrà chiamato il **metodo sort** sulla Collection, automaticamente si sfrutteranno i criteri definiti in Comparable.

Operatore comparator

L'interfaccia **Comparator<T>** consente di delegare il confronto ad una **classe separata**. E' considerata una **scelta obbligatoria** se si vogliono confrontare due oggetti con un **criterio diverso da quello naturale** (rappresentato da Comparable). All'interno della classe separata che implementa questa interfaccia, si deve implementare il metodo **compare(T o1, T o2)**.

```
class NamedPointComparatorByName implements Comparator<NamedPoint> {  
    public int compare(NamedPoint p1, NamedPoint p2) {  
        return p1.getName().compareTo(p2.getName());  
    }  
}
```

fornisce il confronto per nome,
complementa quello naturale (in **Point**) per coordinata

```
class NamedPointComparatorByXY implements Comparator<NamedPoint> {  
    public int compare(NamedPoint p1, NamedPoint p2) {  
        int retval = p1.y - p2.y;  
        if (retval == 0) retval = p1.x - p2.x;  
        return retval;  
    }  
}
```

equivalente al confronto
naturale in **Point**

con generics
(Java 5+)

Per come è strutturato il linguaggio Java, è possibile inserire il criterio di ordinamento di una lista all'interno dei parametri del metodo **sort()** quando

invocato sulla collection in questione: quello che si fa è passare un nuovo oggetto della classe Comparator che implementa solamente il metodo compare:

```
TestCompare() {  
    List<Point> l = new LinkedList<>(  
        // esempi con Comparable  
    );  
    l.clear();  
    l.add(new NamedPoint("B",40,20));  
    l.add(new NamedPoint("D",10,20));  
    l.add(new NamedPoint("C",20,10));  
    l.add(new NamedPoint("A",20,20));  
    System.out.println(l);  
    Collections.sort(l, new NamedPointComparatorByXY());  
    System.out.println(l);  
    l.clear();  
    l.add(new NamedPoint("B",40,20));  
    l.add(new NamedPoint("D",10,20));  
    l.add(new NamedPoint("C",20,10));  
    l.add(new NamedPoint("A",20,20));  
    System.out.println(l);  
    Collections.sort(l, new NamedPointComparatorByName());  
    System.out.println(l);  
}
```

[(40,20), (10,20), (20,10), (20,20)]
[(20,10), (10,20), (20,20), (40,20)]
[(B,40,20), (D,10,20), (C,20,10), (A,20,20)]
[(C,20,10), (D,10,20), (A,20,20), (B,40,20)]

[(B,40,20), (D,10,20), (C,20,10), (A,20,20)]
[(C,20,10), (D,10,20), (A,20,20), (B,40,20)]

[(B,40,20), (D,10,20), (C,20,10), (A,20,20)]
[(A,20,20), (B,40,20), (C,20,10), (D,10,20)]

NOTA: nell'ordinamento XY si dà priorità alla Y maggiore, a parità di Y si confrontano le X, come indicato nell'immagine sopra.