

Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

Capitolo 6: La classe Object: identità, uguaglianza e cast

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Identità e uguaglianza · contains ed equals · La classe Object
Ridefinire equals · Cast e instanceof · Stringhe: == ed equals

Indice

1	Identità e uguaglianza	1
2	contains usa equals	2
3	La classe Object	4
4	Ridefinire equals	5
5	Cast e verifica del tipo	6
6	Stringhe: == ed equals	7
7	Esercizi	8
8	Soluzioni	9

1 Identità e uguaglianza

Definizione – Identità e uguaglianza

Due riferimenti **a** e **b** sono:

- **identici** se riferiscono lo stesso oggetto: lo dice `a == b`;
- **uguali** se riferiscono oggetti che rappresentano la stessa cosa del problema, anche se distinti: lo dice `a.equals(b)`.

Due riferimenti identici sono anche uguali; il contrario non vale. Per vederlo serve una classe. Un punto del piano ha due coordinate, che non cambiano dopo la creazione (oggetto non modificabile, Capitolo 4):

```
public class Punto {  
    private double x;  
    private double y;  
  
    Punto(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    double getX() {  
        return x;  
    }  
  
    double getY() {  
        return y;  
    }  
  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```

Il `toString` scrive il punto nella forma consueta, `(3.0, 2.0)`, senza i nomi dei campi.

```
Punto p = new Punto(0, 0);  
Punto q = new Punto(0, 0);  
Punto r = p;  
// p == q: false    p == r: true
```

Ogni `new` crea un oggetto nuovo: `p` e `q` riferiscono due oggetti distinti con le stesse coordinate, `r` lo stesso oggetto di `p`. Gli oggetti creati da un programma sono tanti quante le `new` eseguite.

Osservazione – Uguali lo decide il problema

Due punti di coordinate (0,0) sono lo stesso punto geometrico, e devono risultare uguali. Due persone con lo stesso nome e cognome possono invece essere persone diverse: per distinguerle serve un dato in più, come il codice fiscale. Se poi l'applicazione rappresenta ogni persona con un solo oggetto, due riferimenti indicano la stessa persona se e solo se sono identici: l'identità basta.

2 contains usa equals**Proposizione – contains e remove**

Per una collezione *c* (insieme o lista) e un riferimento *o* diverso da *null*:

`c.contains(o)` è `true` $\iff$ esiste $e \in c$ con `o.equals(e)`.

Il metodo `remove(o)` di una lista toglie il primo elemento *e* con `o.equals(e)`.

Con `contains`, il metodo che dice se una lista di punti contiene l'origine si scrive in una riga. Il `main` lo prova su una lista senza l'origine e poi con:

```
import java.util.LinkedList;

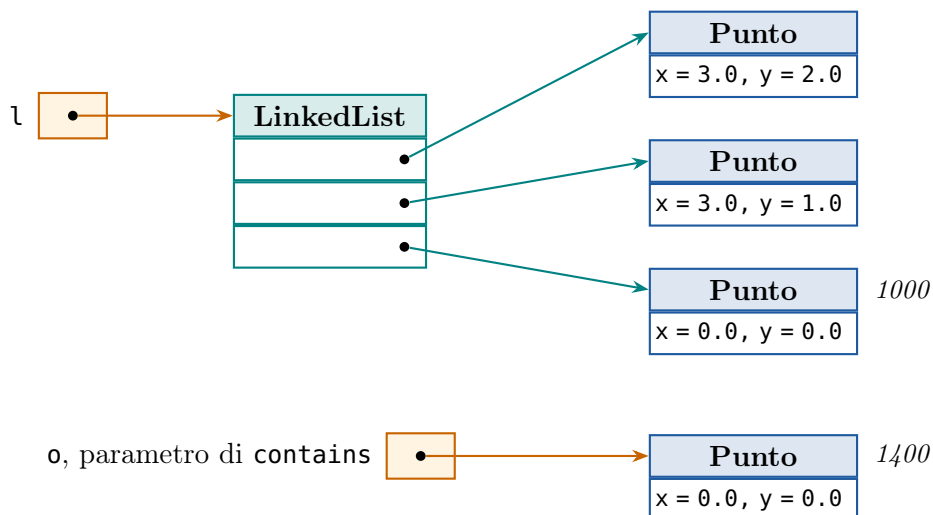
public class Principale {
    static boolean includeOrigine(LinkedList<Punto> l) {
        return l.contains(new Punto(0, 0));
    }

    public static void main(String[] args) {
        LinkedList<Punto> l = new LinkedList<Punto>();
        l.add(new Punto(3, 2));
        l.add(new Punto(3, 1));
        System.out.println(l + " " + includeOrigine(l));
        l.add(new Punto(0, 0));
        System.out.println(l + " " + includeOrigine(l));
    }
}
```

Stampa `false` tutte e due le volte, anche quando l'origine c'è:

```
[(3.0, 2.0), (3.0, 1.0)] false
[(3.0, 2.0), (3.0, 1.0), (0.0, 0.0)] false
```

Il punto aggiunto con `l.add(new Punto(0, 0))` e quello che `includeOrigine` passa a `contains` sono due oggetti: uguali, ma non identici. La classe `Punto` non dice quando due punti sono uguali, e il suo `equals` è quello predefinito (Sezione 3), che confronta i riferimenti come `==`. Così `contains` cerca proprio l'oggetto ricevuto, che nella lista non c'è.



Alla seconda chiamata la lista contiene il riferimento al punto 1000, `contains` riceve il punto 1400: stesse coordinate, oggetti diversi. Con l'`equals` predefinito il confronto dà `false`.

Con lo stesso oggetto, invece, `contains` lo trova: dopo `Punto p = new Punto(0, 0); l.add(p);`, `l.contains(p)` è `true`. Perché il `main` stampi `true` alla seconda riga, `Punto` deve dire quando due punti sono uguali (Sezione 4). Esercizio 1.

Osservazione – Metodi chiamati da altri

Costruttore, `toString` ed `equals` si scrivono perché li chiami qualcun altro, non il programma direttamente:

Metodo	Chi lo chiama
costruttore	<code>new</code> , dopo aver creato l'oggetto
<code>toString</code>	<code>System.out.println</code>
<code>equals</code>	<code>contains</code> e <code>remove</code> delle collezioni

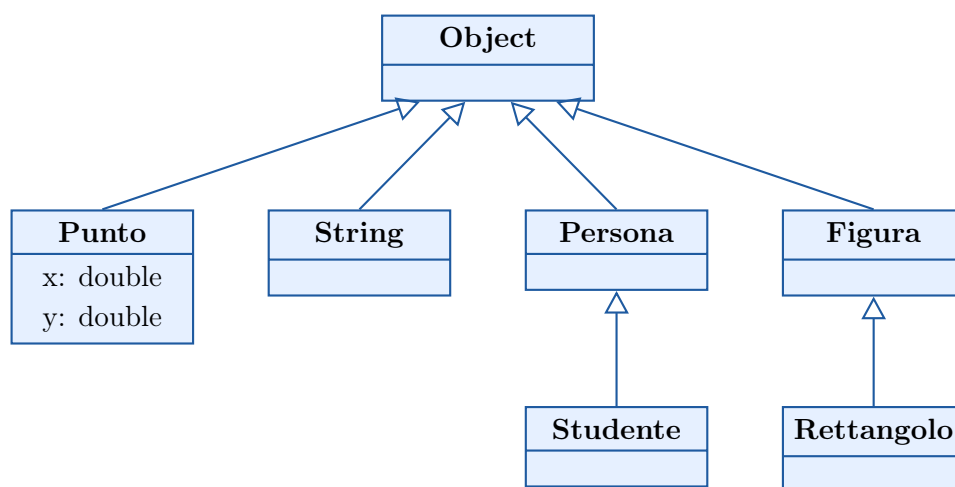
Chi chiama si aspetta un'intestazione precisa: un metodo con un'altra intestazione non viene chiamato.

3 La classe Object

Definizione – Object

Object è la radice della gerarchia delle classi Java: una classe che non dichiara `extends` è sottoclasse di **Object**. Scrivere `class Punto { ... }` equivale a scrivere `class Punto extends Object { ... }`.

Ogni classe è quindi sottoclasse di **Object**, direttamente o attraverso altre classi, e ne eredita i metodi, fra cui `toString` ed `equals`. Il `toString` predefinito del Capitolo 3, che stampa `Figura@3d4eac69`, è quello di **Object**.



*Alcune classi dei capitoli precedenti sotto **Object**. **Studente** e **Rettangolo** ne sono sottoclassi attraverso **Persona** e **Figura**.*

L'equals di Object. Nella libreria Java è scritto così:

```

public boolean equals(Object obj) {
    return (this == obj);
}
  
```

In **Object**, quindi, uguaglianza e identità coincidono. Le collezioni accettano oggetti di qualsiasi classe, e il loro `contains` chiama questo `equals`, a meno che la classe degli elementi non lo ridefinisca.

Variabili di tipo Object. Per la compatibilità dei tipi (Capitolo 3), una variabile di tipo **Object** può riferire un oggetto di qualsiasi classe. Il compilatore però guarda il tipo della variabile: si possono chiamare solo i metodi di **Object**.

```

Object o = new Punto(0, 0);
System.out.println(o);    // (0.0, 0.0): si esegue il toString di Punto
o.getX();                 // errore di compilazione
  
```

L'ultima riga non compila (*cannot find symbol*): non tutti gli oggetti hanno una coordinata x .

4 Ridefinire equals

Proposizione – Ridefinizione di equals

Per stabilire quando due oggetti di una classe sono uguali si ridefinisce **equals**:

- l'intestazione è esattamente `public boolean equals(Object o);`
- il corpo restituisce **false** se **o** non è un oggetto della classe, altrimenti confronta i campi che decidono l'uguaglianza.

```
// in Punto
public boolean equals(Object o) {
    if (!(o instanceof Punto)) {
        return false;           // o è null, o non è un punto
    }
    Punto p = (Punto) o;       // cast: Sezione 5
    return x == p.x && y == p.y;
}
```

Il parametro è di tipo **Object**, quindi `o.x` non compila. Dopo la verifica, il **cast** (**Punto**) **o** mette lo stesso riferimento in una variabile di tipo **Punto**, e da lì le coordinate si leggono. `p.x` si può scrivere anche se `x` è **private**: siamo dentro **Punto** (Capitolo 4).

Con questo **equals** il **main** della Sezione 2 stampa **false** e poi **true**. Per ogni elemento **e** della lista, **contains** chiama **equals** sul punto ricevuto, con **e** come argomento; arrivato al punto (0,0), le coordinate coincidono e il risultato è **true**. Anche `l.remove(new Punto(3, 1))` ora toglie il punto (3,1).

Osservazione – Errori tipici

- **Parametro di tipo Punto.** `public boolean equals(Punto p)` è un altro metodo, con lo stesso nome e un parametro diverso: non ridefinisce quello di **Object**, e **contains** continua a dare **false**.
- **Campi scelti male.** I campi confrontati decidono l'uguaglianza: confrontando solo `x`, (0,0) e (0,5) sarebbero lo stesso punto.

Nota – Con HashSet serve anche hashCode

Un `HashSet` usa, oltre a `equals`, il metodo `hashCode` di `Object`, che va ridefinito in modo coerente: oggetti uguali devono avere lo stesso `hashCode`. Con il solo `equals`, se `h` è un `HashSet<Punto>` che contiene `(0,0)`, `h.contains(new Punto(0, 0))` dà ancora `false`. Per questo gli esempi del capitolo usano una `LinkedList`.

5 Cast e verifica del tipo

Definizione – Cast

Se `e` è un riferimento e `C` una classe, `(C) e` è lo stesso riferimento, visto come tipo `C`.

- Non crea né converte l'oggetto: cambia solo il tipo con cui lo vede il compilatore.
- Compila solo se `C` e il tipo di `e` sono una sottoclasse dell'altro.
- Durante l'esecuzione, se l'oggetto non è un `C`, lancia una `ClassCastException`.

```
Object o = "ciao";  
Punto z = (Punto) o;           // compila, ma lancia ClassCastException  
Punto w = (Punto) "ciao";     // errore di compilazione
```

La seconda riga compila, perché una variabile `Object` potrebbe riferire un punto; durante l'esecuzione l'oggetto si rivela una stringa, e il programma si ferma con *class java.lang.String cannot be cast to class Punto*. La terza non compila (*incompatible types: String cannot be converted to Punto*): una `String` non è mai un punto, e il compilatore lo sa già.

Definizione – instanceof

L'espressione `o instanceof C` è `true` se `o` non è `null` e l'oggetto riferito è di classe `C` o di una sua sottoclasse: esattamente quando il cast `(C) o` riesce.

Con gli insiemi del Capitolo 5: una variabile `Object` può riferire un oggetto dovunque; il cast `(Punto) o` afferma che l'oggetto sta nel sottoinsieme `Punto`, e `o instanceof Punto` lo verifica prima di affermarlo. Per questo l'`equals` della Sezione 4 restituisce `false` invece di fermarsi: `p.equals("ciao")` e `p.equals(null)` danno `false`.

6 Stringhe: == ed equals

Proposizione – Confrontare stringhe

`String` è una classe, e `==` fra stringhe confronta i riferimenti.

- Il contenuto si confronta con `equals`: `s.equals("")`.
- `null` si confronta con `==`, e prima di chiamare `equals`.

```
String a = "";
String b = new String("");
System.out.println(a == "");           // true
System.out.println(b == "");           // false
System.out.println(b.equals(""));      // true
```

Due letterali uguali nel codice sono lo stesso oggetto, quindi `a == ""` è `true`. `new String("")` crea invece un'altra stringa vuota, uguale ma non identica. Un controllo scritto con `==` funziona con i letterali e sbaglia con le stringhe costruite durante l'esecuzione, come quelle lette dall'utente.

Nel costruttore di una persona, il controllo che rifiuta un nome mancante è:

```
if (nome == null || nome.equals("")) {
    throw new RuntimeException("Nome mancante");
}
```

L'ordine conta: l'operatore `||` non valuta la seconda condizione se la prima è già vera. Invertendo le due condizioni, un nome uguale a `null` farebbe fallire la chiamata di `equals` con una `NullPointerException`.

Riepilogo

- **Identità:** `a == b`, stesso oggetto; ogni `new` crea un oggetto distinto.
- **Uguaglianza:** `a.equals(b)`, la stessa cosa del problema; quando lo sono lo decide il problema.
- **contains e remove:** cercano un elemento `e` con `o.equals(e)`.
- **Object:** radice della gerarchia; il suo `equals` è `this == obj`.
- **Ridefinire equals:** `public boolean equals(Object o)`, verifica del tipo, cast, confronto dei campi.
- **HashSet:** oltre a `equals` serve `hashCode` coerente.
- **Cast:** `(C) e` è lo stesso oggetto visto come `C`; `ClassCastException` se non lo è, verifica con `instanceof`.
- **Stringhe:** contenuto con `equals`, `null` con `==` e per primo.

7 Esercizi

Esercizio 1 – L'origine in una lista di punti

Con la classe **Punto** della Sezione 1, senza **equals**, scrivere un metodo **static boolean includeOrigine(LinkedList<Punto> l)** che restituisca **true** se la lista contiene il punto (0,0).

- (a) Scriverlo con un ciclo for-each.
- (b) Scriverlo con **contains**. Che cosa va aggiunto a **Punto**?
- (c) Con quali dati va provato?

Soluzione a pagina 9 →

8 Soluzioni

Soluzione dell'Esercizio 1 – L'origine in una lista di punti

(a) Si scorre la lista confrontando le coordinate:

```
static boolean includeOrigine(LinkedList<Punto> l) {  
    for (Punto p : l) {  
        if (p.getX() == 0 && p.getY() == 0) {  
            return true;  
        }  
    }  
    return false;  
}
```

Il ciclo confronta i valori, non i riferimenti, e funziona anche senza `equals`. Con il `main` della Sezione 2 stampa `false` e poi `true`. Lo stesso metodo funziona con un `HashSet<Punto>`.

(b) `return l.contains(new Punto(0, 0));` cerca un elemento `e` con `new Punto(0, 0).equals(e)`. Con l'`equals` di `Object` il punto appena creato non è identico a nessun elemento, e il risultato è sempre `false`. Va aggiunto a `Punto` l'`equals` della Sezione 4, che confronta le coordinate: allora il `main` stampa `false` e poi `true`. Con un `HashSet` servirebbe anche `hashCode`.

(c) Almeno una lista senza l'origine, dove il risultato atteso è `false`, e una con l'origine, dove è `true`. Una prova sola non basta: senza `equals`, la versione con `contains` supera la prova senza origine e sbaglia l'altra. Il `main` della Sezione 2 fa entrambe le prove.

[← Torna all'Esercizio 1, pagina 8](#)