

Appunti di Programmazione Concorrente e Distribuita

(A.A 2018/2019)

Grigoras Valentin

Matricola: 1099561

Contenuti

1	Metodo <i>Equals()</i>	4
1.1	Proprietà del metodo <i>Equals()</i>	4
1.2	Corretta implementazione di <i>Equals()</i>	4
1.3	Binding dinamico di <i>Equals()</i>	5
1.4	Metodo <i>Equals()</i> ed eredità	6
1.5	Tipi primitivi vs oggetti	6
1.6	<i>Equals()</i> Vs <i>==</i>	8
2	Operatore <i>instanceof</i>	9
3	Il metodo <i>String toString()</i>	10
3.1	Costruzione stringhe	10
3.2	Concatenazione stringhe	10
4	Le classi wrapper	11
4.1	Autoboxing - Autounboxing	11
4.2	Cose da evitare	11
5	Il metodo <i>Clone()</i>	12
5.1	<i>Clone()</i> con copia profonda	13
6	La keyword <i>final</i>	13
6.1	Campi dati <i>final</i>	14
6.2	Metodi <i>final</i>	15
6.3	Classi <i>final</i>	15

Elenco delle figure

1	Implementazione corretta di equals	5
2	tipi primitivi e oggetti: rappresentazione in memoria	7
3	tipi primitivi e oggetti: rappresentazione in memoria dopo assegnazione	8
4	Shallow copy vs Deep copy	12
5	Deep copy Clone()	13

1 Metodo Equals()

L'operatore binario `==` viene utilizzato per stabilire se il contenuto di due variabili è identico. Per i tipi non primitivi viene confrontato il contenuto dei puntatori degli oggetti, ovvero l'indirizzo di memoria degli oggetti a cui i puntatori fanno riferimento. Spesso però potremmo avere oggetti che nonostante puntino ad indirizzi di memoria differenti possono essere considerati a tutti gli effetti uguali. La procedura standard per confrontare oggetti in Java è il metodo `equals()`, definito in **Object**, quindi è disponibile su tutti gli oggetti (ma non sui tipi primitivi):

```
/*segnatura*/
public boolean equals(Object x)

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

Il confronto viene fatto con l'operatore `==`, dato che non è possibile conoscere in **Object** come confrontare qualsiasi oggetto Java. Per effettuare un confronto migliore non basato sugli indirizzi di memoria, è buona norma effettuare l'overriding del metodo `equals` in ogni classe che sviluppiamo.

1.1 Proprietà del metodo Equals()

Il linguaggio Java richiede che qualunque ridefinizione del metodo `equals()` rispetti determinate proprietà che forniscono una vera e propria **relazione di equivalenza**:

- **Riflessività**: per ogni oggetto `x`, avremo che `x.equals(x)` ritorna `true`;
- **Simmetria**: dati due oggetti `x` ed `y`, avremo che se `x.equals(y)` ritorna `true`, allora `y.equals(x)` ritorna `true`;
- **Transitività**: dati tre oggetti `x`, `y` e `z`, se `x.equals(y)` ritorna `true` e `y.equals(z)` ritorna `true`, allora `x.equals(z)` ritorna `true`.

1.2 Corretta implementazione di Equals()

Si devono eseguire 3 step nella classe più un quarto di verifica all'esterno:

1. Usare `==` per vedere se l'argomento è uguale a `this`;

```
if (this == obj) return true;
```

2. Verificare se l'oggetto passato al metodo è del tipo desiderato, effettuando un controllo con l'operatore `instanceof`, in caso negativo restituiremo `false`;

```
if (!(obj instanceof ClassName)) return false;
```

3. Essendo sicuri di avere un oggetto del tipo desiderato, potremo effettuare in sicurezza un downcast al tipo desiderato. In questo modo possiamo accedere ai campi e metodi prima nascosti a causa del riferimento di tipo **Object** utilizzato;

```

NomeClasse cp = (NomeClasse) obj;
return cp.campo1.equals(this.campo1) &&
       cp.campoN.equals(this.campoN);

```

4. Confrontare se stiamo rispettando la riflessibilità, simmetria e transitività.

```

public class ColorPoint {
    private Point point;
    private Color color;

    public ColorPoint(...) {...}

    public Point asPoint() {
        return new Point(point.getX(), point.getY());
    }

    @Override
    public boolean equals(Object obj) {
        if(this == obj) return true;
        if(!(obj instanceof ColorPoint)) return false;
        ColorPoint cp = (ColorPoint)obj;
        return cp.point.equals(point) &&
               cp.color.equals(color);
    }
}

```

Figura 1: Implementazione corretta di equals

1.3 Binding dinamico di Equals()

Quando si effettua l'overriding di equals, è sconsigliato cambiare il tipo del parametro di equals Object, altrimenti non avremo un overriding ma un overloading. Questo cambiamento potrebbe provocare disastri, vediamo con un esempio in cui cambiando il tipo del parametro di equals cosa potrebbe accadere:

```

public boolean equals(Employee e) {}

```

Inoltre supponeremo di avere:

```

Employee e1 = new Employee (...);
Employee e2 = new Employee (...);
Object o1 = e1;
Object o2 = e1;

```

Analizziamo cosa succede ad ogni riga con l'utilizzo del Binding dinamico:

- **o1.equals(o2);** Le firme candidate sono cercate solo in Object perché il tipo dichiarato di o1 è Object. Di conseguenza non verrà considerata la specializzazione del metodo equals effettuata;

- ***o1.equals(e2)***; Nella fase 2 del Binding dinamico non c'è overriding, quindi verrà preso l'unico metodo equals esattamente uguale che sarà quello contenuto in *Object*. Anche in questo caso non verrà considerata la specializzazione del metodo equals che abbiamo effettuato;
- ***e1.equals(o1)***; Viene valutato ma *Object* non è assegnabile a *Employee*. Di conseguenza non verrà considerata la specializzazione del metodo equals effettuata;
- ***e1.equals(e2)***; La firma combacia con quella specializzata e sarà l'unico caso in cui verrà chiamato il "giusto" equals.

1.4 Metodo *Equals()* ed eredità

Quando il metodo equals viene sovrascritto bisogna prestare attenzione al funzionamento tra il metodo equals definito e il suo funzionamento nelle classi sottostanti. In definitiva bisogna decidere come si deve comportare il metodo equals con le sue sottoclassi. In particolare, bisogna porsi due domande:

1. **Il criterio di confronto tra oggetti di una sottoclasse è diverso da quello che vale tra oggetti della superclasse?**
 - No, allora bisogna definire equals della superclasse final in modo che nessuna sottoclasse possa sovrascriverlo;
 - Sì, allora il metodo equals deve essere opportunamente ridefinito in ogni sottoclasse in cui il criterio di confronto è differente.
2. **Un oggetto di una sottoclasse può essere considerato uguale ad un oggetto di una superclasse?**
 - No, allora bisogna verificare di ammettere il confronto solo tra agli oggetti del tipo giusto;
 - Sì, allora se gli oggetti di una sottoclasse possono essere considerati uguali a quelli di una superclasse dovremo ammettere al confronto tutti gli oggetti di quel tipo e tutte le sue sottoclassi.

1.5 Tipi primitivi vs oggetti

Si sente dire spesso l'affermazione *in Java tutto è un oggetto*. In realtà questa frase non è del tutto vera in quanto esistono i tipi primitivi che permettono di utilizzare numeri, caratteri e valori booleani senza ricorrere agli oggetti. I tipi primitivi sono riconoscibili anche perché iniziano con una lettera minuscola (mentre i nomi delle classi iniziano sempre con una maiuscola). I tipi primitivi disponibili in Java sono i seguenti:

- **boolean**: valore booleano, può assumere i valori true e false;
- **byte**: numero intero a 8 bit;
- **short**: numero intero a 16 bit;
- **int**: numero intero a 32 bit;
- **long**: numero intero a 64 bit;
- **float**: numero reale a 32 bit in virgola mobile (IEEE 754-1985);

- **double:** numero reale a 64 bit in virgola mobile (IEEE 754-1985);
- **char:** carattere unicode a 16 bit;

Gli oggetti in Java sono istanze di una classe. Una delle differenze importanti fra tipi primitivi e oggetti si ha se guardiamo la situazione della memoria a basso livello. Infatti nella locazione di memoria corrispondente a un tipo primitivo è presente il valore mentre in quella di un oggetto è presente il puntatore all'area di memoria che contiene l'oggetto. Un diagramma spiega meglio questa differenza:

```
int i1 = 5;  
int i2 = 7;  
Persona p1 = new Persona("Mario", "Rossi");  
Persona p2 = new Persona("Mario", "Verdi");
```

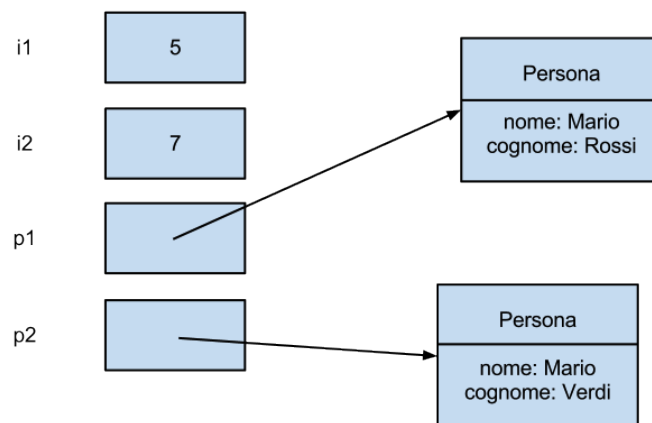


Figura 2: tipi primitivi e oggetti: rappresentazione in memoria

Fin qui sembra tutto semplice, andiamo avanti facendo un po' di assegnazioni:

```
i2 = i1;  
p2 = p1;
```

A questo punto quale è la situazione? Ovviamente `i1` e `i2` assumono lo stesso valore, stessa cosa anche per `p1` e `p2`. Ma se andiamo a vedere cosa è successo in memoria la situazione è un po' più complicata:

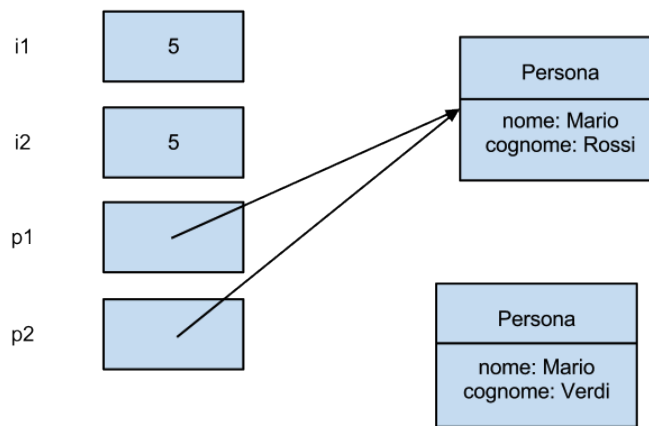


Figura 3: tipi primitivi e oggetti: rappresentazione in memoria dopo assegnazione

Adesso p1 e p2 puntano allo stesso oggetto: abbiamo una condivisione di memoria che può risultare pericolosa se non gestita adeguatamente. Infatti richiamando un metodo setter su uno dei due oggetti (per esempio `p1.setNome("Fabio")`) si modificherà l'oggetto condiviso dai due puntatori.

1.6 *Equals()* Vs *==*

Iniziamo con l'operatore *==*: invocando questo costrutto viene confrontato il valore contenuto nella variabile. Quindi, nel caso di tipi primitivi viene confrontato il valore vero, mentre nel caso di oggetti viene confrontato l'indirizzo di memoria a cui i puntatori fanno riferimento. Vediamo un esempio concreto. Il seguente codice stampa sulla console 4 volte il valore true:

```
int i1 = 5;
int i2 = 7;

System.out.println(i1 != i2);
i2 = i1;
System.out.println(i1 == i2);

Persona p1 = new Persona("Mario", "Rossi");
Persona p2 = new Persona("Mario", "Verdi");

System.out.println(p1 != p2);
p2 = p1;
System.out.println(p1 == p2);
```

p1 e p2 puntano allo stesso oggetto e quindi confrontando l'indirizzo di memoria corrispondente al puntatore si ha un esito positivo. Vediamo un altro esempio leggermente più complicato. Definiamo due volte lo stesso valore/oggetto e confrontiamolo con *==*:

```
int i1 = 5;
int i2 = 5;

System.out.println(i1 == i2);
```



```
Persona p1 = new Persona("Mario", "Rossi");
Persona p2 = new Persona("Mario", "Rossi");

System.out.println(p1 == p2);
```

In questo caso la prima condizione sui tipi primitivi è vera mentre quella sugli oggetti è falsa! Il motivo è chiaro, le variabili p1 e p2 in questo caso puntano a locazioni di memoria diverse quindi confrontando gli indirizzi di memoria si avrà un risultato negativo. Per ovviare a questo problema è necessario usare il metodo equals che, a differenza dell'operatore ==, confronta gli oggetti puntati entrando quindi in merito al contenuto dell'oggetto. Riproviamo quindi a eseguire lo stesso codice usando il metodo equals per confrontare i due oggetti:

```
Persona p1 = new Persona("Mario", "Rossi");
Persona p2 = new Persona("Mario", "Rossi");

System.out.println(p1.equals(p2));
```

Eseguendo questo codice viene stampato sulla console ancora false! Come mai? Il motivo è che il metodo equals è definito dentro la classe Object con la seguente implementazione:

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

L'implementazione di Object del metodo equals utilizza l'operatore ==! Questa scelta può sembrare strana ma è l'unica che poteva essere fatta. Infatti la classe Object è la classe padre di tutte le classi Java, non è possibile dentro questa classe sapere come confrontare qualunque oggetto Java!

2 Operatore instanceof

È un operatore che consente di determinare a run-time il tipo di un oggetto. Restituisce true o false a seconda che l'oggetto sia o no un'istanza della classe di confronto o di una delle sue superclassi. L'operatore si applica anche alle interfacce. Se un oggetto implementa un'interfaccia allora instanceof con il nome di quell'interfaccia restituisce true. La sintassi prevede la seguente dichiarazione:

```
<referimento> instanceof <tipo_referimento>
```

Esempi:

```
/*esempio 1*/
if ("prova" instanceof String) ... // restituisce true

/*esempio 2*/
Point pt = new Point(3,5);
if (pt instanceof String) ... // restituisce false
```

3 Il metodo String toString()

Si tratta di un metodo che restituisce una rappresentazione dell'oggetto di invocazione in formato stringa. L'implementazione di default restituisce una stringa contenente il nome della classe e l'indirizzo di riferimento di invocazione:

```
<classe>@<hashcode>
```

dove *classe* è il nome della classe dell'oggetto su cui il metodo è invocato, e *hashcode* è la rappresentazione esadecimale del codice hash dell'oggetto (indirizzo in memoria dell'oggetto). Questo accade perché la classe Object non può conoscere la struttura dell'oggetto. Invocazioni implicite del metodo *toString* sono inoltre inserite dal compilatore in ogni contesto dove sia usato un riferimento e si richiede invece un valore di tipo String, ad esempio il codice:

```
Data d = new Data();
System.out.println(d);
```

è tradotto dal compilatore in *System.out.println(d.toString());* chiamando cioè sul riferimento *d* il metodo *toString()* che traduce l'oggetto di tipo *Data* in una stringa, che può essere passata come parametro al metodo *println*.

3.1 Costruzione stringhe

Dato il seguente codice :

```
String nome= new String(''Roberto''); /*1*/
String nome = ''Roberto''; /*2*/
```

è importante ricordare che l'operazione più efficiente a lungo andare è la seconda. Nel primo caso ho un reference perché alloco nello heap un oggetto, nel secondo caso NON ho un reference e costruisco la stringa che è costante e so dove si trova in memoria.

3.2 Concatenazione stringhe

Dato il seguente codice:

```
String nomi = '' '';
for(var p : people)
    nomi += p;

var nomi= new StringBuilder('' ');
for(var p : people)
    nomi.append(p);
```

è importante ricordare che il tipo String è immutabile e ogni volta che si fa il += viene allocata una nuova stringa, quindi chiamare += spesso causa un continuo ri-alloco di stringhe. Lo StringBuilder invece viene allocato una volta sola (al momento di creazione, col new) e poi con l'append si aggiunge testo in coda senza le riallocazioni (come accadeva prima).

4 Le classi wrapper

Sono dette **classi wrapper** (involucro) le classi che fanno da contenitore ad un tipo di dato primitivo, astruendo proprio il concetto di tipo primitivo. Le classi wrapper sono le seguenti: *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Booleab*, *Character*, ognuna delle quali può contenere il relativo tipo primitivo. Tutte le class wrapper appartene la Boolean sono sottoclassi di *Number* Come la classe *String*, anche le classi wrapper sono immutabili. Questo implica che un oggetto istanziato da una classe wrapper non potrà mai cambiare il suo valore interno tramite un suo metodo, anche perchè non esistono metodi setter. Quindi un oggetto istanziato così:

```
Boolean b = new Boolean(true);
```

non conterrà mai il valore false. Le classi wrapper sono utili soprattutto nei casi in cui dobbiamo utilizzare un tipo di dato primitivo laddove è richiesto un oggetto:

```
/*Errore! I tipi generici richiedono tipi parametro
solo complessi e non primitivi*/
ArrayList<int> list = new ArrayList<>();

/*sfrutta la regola dell'autoboxing-autounboxing*/
ArrayList<Integer> list ) = new ArrayList<>();
```

4.1 Autoboxing - Autounboxing

Autoboxing: conversione automatica (fatta dal compilatore) dei tipi primitivi nei rispettivi wrapper; **Autounboxing**: conversione automatica (fatta dal compilatore) dei tipi wrapper nei rispettivi tipi primitivi;

Esempio:

```
/* legale perche c é una coincidenza perfetta tra tipo
primitivo e wrapper*/
Double d = 2.2;

Double d = 2; /*illegale perché 2 é intero*/
```

4.2 Cose da evitare

Il seguente codice compila ma è inefficiente:

```
Long sum = 0L;
var max = Integer.MAX_VALUE;
for(long i = 0; i < max; ++i)
    sum += i;
```

Il tipo di sum è Long, ovvero un wrapper di long. Nel for invece ho longche è un tipo primitivo e quindi la chiamata a sum += i;effettua il boxing del tipo, quindi crea conversioni di continuo che impattano sulla performance del programma.

Soluzione: usare long sum;

5 Il metodo Clone()

Il metodo *clone()* restituisce un nuovo oggetto il cui stato iniziale è una copia dell'oggetto su cui viene invocato.

Se si vuole che sia possibile copiare gli oggetti di una classe, la classe deve implementare l'interfaccia *Cloneable*. Il metodo *clone()* che viene ereditato dalla classe *Object* controlla prima di tutto che la classe dell'oggetto implementi l'interfaccia *Cloneable* e in caso contrario lancia l'eccezione *CloneNotSupportedException*, altrimenti crea un nuovo oggetto e lo inizializza con una copia degli attributi dell'oggetto originale; al termine restituisce un riferimento al nuovo oggetto.

Per garantire che il metodo *Clone()* non sia invocato accidentalmente, il metodo è dichiarato *protected*. Essendo dichiarato *protected*, per essere reso disponibile per l'invocazione nelle nostre classi bisognerà sottoporlo a *override* cambiandone il modificatore a *public*, come nel seguente esempio:

```
public class Quadrato implements Cloneable {
    ...
    public Object clone () {
        try {
            Object ogg = super.clone ();
            return ogg;
        }
        catch (CloneNotSupportedException e ) {
            return null;
        }
    }
}
```

È importante notare che la copia che viene fatta dal metodo *Clone()* della classe *Object*, si limita a copiare i valori delle variabili d'istanza dell'oggetto (si tratta di shallow copy o copia superficiale). Quindi se le variabili d'istanza sono reference ad altri oggetti, allora verranno copiati i loro indirizzi, e non il contenuto del reference creandosi così una condivisione di memoria.

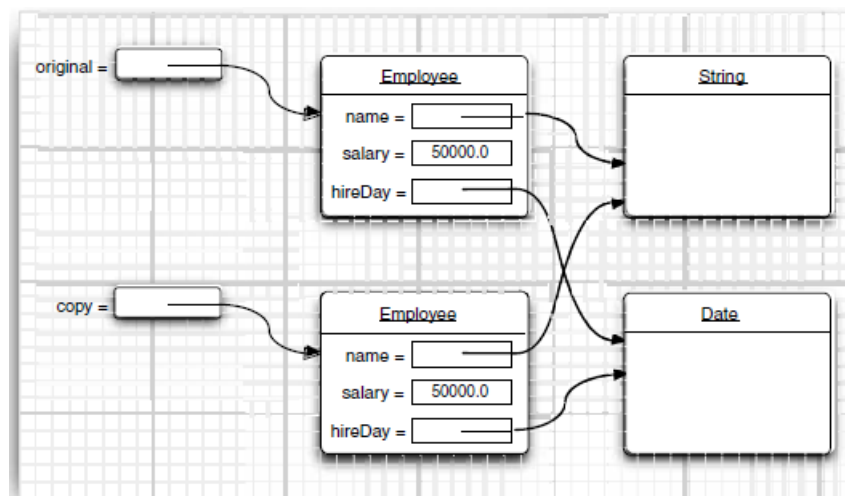


Figura 4: Shallow copy vs Deep copy

5.1 Clone() con copia profonda

```
public class Stack implements Cloneable {  
    private Object[] elements;  
  
    @Override public Stack clone() {  
        try {  
            Stack result = (Stack)super.clone();  
            result.elements = elements.clone();  
        } catch (CloneNotSupportedException cnsex)  
        {  
            throw new AssertionError();  
        }  
        return result;  
    }  
}
```

Figura 5: Deep copy Clone()

1. chiamare il metodo clone della superclasse;
2. castare il risultato al tipo stesso della classe;
3. si fa una copia profonda dei campi che hanno delle reference mutabili. Notare che su `elements` è stato chiamato `clone()` perché è il metodo che fa una copia profonda di un array di `Object` ma volendo si sarebbe potuto fare manualmente un `for` e fare deep copy del contenuto dell'array;
4. i ritorna il risultato.

La `clone()` comunque potrebbe essere un problema se un campo è `final`, in generale è meglio avere un metodo di cloning separato e non implementare `Cloneable`. Ad esempio, si può usare un costruttore di copia oppure un factory method, ovvero un metodo che costruisce per me l'oggetto. Esempio di factory:

```
public static Stack factoryNew(Stack source) {  
    Stack dest = new Stack (...);  
    //faccio copie deep o altro prendendo i dati da source  
    return dest;  
}
```

L'idea del factory è: creo un metodo da usare tipo `Stack copia = Stack.factoryNew(...)` che ritorna un oggetto di tipo `Stack` che è una copia esatta dell'oggetto che passo in input.

6 La keyword final

La keyword *final* si può utilizzare in contesti diversi ma in generale identifica qualcosa di costante o immutabile. Il reference this, che in un metodo non statico si riferi-

sce all'oggetto di invocazione è sempre implicitamente *final*, ovvero non modificabile. Invece l'oggetto a cui *this* si riferisce è modificabile:

```
public class C {  
    public int x;  
    public void metodo(){  
        this = new C(); /* ILLEGALE: cannot assign a  
        value to final variable this*/  
        this.x = 10; //ok  
    }  
}
```

6.1 Campi dati final

Un campo dati statico o no marcato *final* è un campo costante / immutabile, quindi ad ogni tentativo di modifica il compilatore segna un errore. I campi dati marcati *final* devono essere sempre marcati esplicitamente, non basta l'inizializzazione automatica:

```
public class E {  
    final int i; // ILLEGALE  
}  
public class E {  
    static final int k; // ILLEGALE  
    static final int v=0; //ok  
}  
public class E {  
    final int i;  
    E(){i=0;} // ok  
}  
public class E {  
    final int i;  
    E(){i=0; i=2;} /* ILLEGALE, non posso modificare  
    il valore di i dopo avergli assegnato 0*/  
}
```

Se un riferimento *ref* viene marcato *final*, allora *ref* si riferisce ad uno ed un solo oggetto fissato, ma il contenuto dell'oggetto puntato può essere modificato. Java non fornisce un modo per rendere costanti gli oggetti:

```
class Z{  
    int i=2;  
}  
  
public class D{  
    Z z1 = new Z();  
    final Z z2 = new Z();  
    static final Z z3 = new Z();  
    final int[] a = {1, 2, 3, 4, 5};  
  
    public static void main(String[] args){  
        D d1 = new D();  
    }  
}
```

```
d1.z2.i++; //ok
d1.z1 = new Z();
for(int i = 0; i < d1.a.lenght; i++){
d1.a[i]++; /* ok, solamente il reference a é costante
d1.z2 = new Z(); ILLEGALE, z2 é final
D.z3 = new Z(); ILLEGALE, z3 é final
d1.a = new int[3]; ILLEGALE, a é final
}
}
}
```

6.2 Metodi final

Un metodo marcato come *final* indica che tale metodo non può essere ridefinito in eventuali sottoclassi. I motivi per marcare un metodo final sono:

- **Sicurezza:** si può pensare che un metodo *boolean validatePassword()* può essere marcato final per evitare eventuali ridefinizioni maliziose;
- **Efficienza;**
- **Progettuali:** a volte si desidera mettere un blocco sul metodo per evitare che una sottoclasse cambi comportamento.

```
public class Classe {
    //attributi
    public final void MetodoNonRidefinibile() {
        //codice del metodo non ridefinibile
    }
    //codice
}
```

6.3 Classi final

Marcare una classe C come final significa che non possono essere definiti sottoclassi di C. Naturalmente, tutti i metodi di una classe final sono implicitamente final, mentre i campi dati possono o meno essere marcati final. Si noti che marcare una classe final si opera una notevole restrizione a suo utilizzo in quanto non sarebbe permesso l'estendibilità di quella classe.