

# RIUSABILITÀ

- Si vuole riusare tutto ciò che può essere riusato (*componenti, codice, astrazioni*)
- Non è utile né opportuno modificare codice *già funzionante e corretto*
  - il cui sviluppo ha richiesto tempo (anni-uomo)
  - ed è costato (molto) denaro
- Occorre disporre nel linguaggio di un modo per progettare alle differenze, procedendo *in modo incrementale*

Ereditarietà - 1

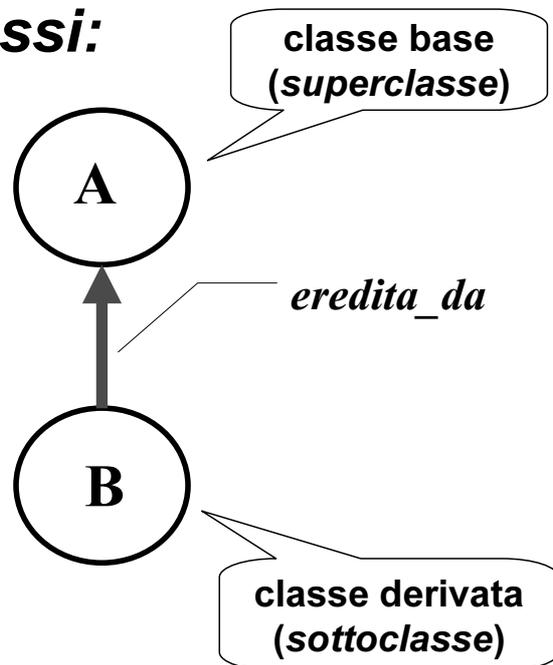
# L'OBIETTIVO

- Poter definire una nuova classe a partire da una già esistente
- Bisognerà dire:
  - quali dati la nuova classe *ha in più* rispetto alla precedente
  - quali metodi la nuova classe *ha in più* rispetto alla precedente
  - quali metodi la nuova classe *modifica* rispetto alla precedente

Ereditarietà - 2

# EREDITARIETÀ

- Una *relazione tra classi*: si dice che la nuova classe B eredita dalla preesistente classe A



Ereditarietà - 3

# EREDITARIETÀ

- La nuova classe **ESTENDE** una classe già esistente
  - può aggiungere nuovi dati o metodi
  - può accedere ai dati ereditati purché il livello di protezione lo consenta
  - non può eliminare dati o metodi !!
- La classe derivata condivide *la struttura e il comportamento* (per le parti non ridefinite) della classe base

Ereditarietà - 4

## ESEMPIO

Dal contatore (solo in avanti) ...

```
public class Counter {  
    private int val;  
    public Counter()      { val = 1; }  
    public Counter(int v) { val = v; }  
    public void reset()   { val = 0; }  
    public void inc()     { val++; }  
    public int getValue() { return val; }  
}
```

Attenzione alla protezione!

Ereditarietà - 5

## ESEMPIO

*... al contatore avanti/indietro (con decremento)*

```
public class Counter2 extends Counter {  
    public void dec() { val--; }  
}
```

**Questa nuova classe:**

- eredita da Counter il campo `val` (un `int`)
- eredita da Counter *tutti i metodi*
- aggiunge a Counter il metodo `dec()`

Ereditarietà - 6

## ESEMPIO

*... al contatore avanti/indietro (con decremento)*

```
public class Counter2 extends Counter {  
    public void dec() { val--; }  
}
```

Que

Ma val era privato!!

• er

**ERRORE: nessuno può** (nt)

• er

accedere a dati e metodi

• ag

**privati di qualcun altro!!!**

Ereditarietà - 7

## EREDITARIETÀ E PROTEZIONE

- **Problema**: il livello di protezione `private` impedisce a chiunque di accedere al dato, *anche a una classe derivata*
  - va bene per dati “veramente privati”
  - ma è *troppo restrittivo* nella maggioranza dei casi
- Per sfruttare appieno l’ereditarietà occorre *rilassare un po’ il livello di protezione*
  - senza dover tornare però a `public`
  - senza dover scegliere per forza la protezione `package` di default: il concetto di `package` *non c’entra niente* con l’ereditarietà!

Ereditarietà - 8

# LA QUALIFICA `protected`

## Un dato o un metodo `protected`

- è come `package` (il default) per chiunque non sia una classe derivata
- ma *consente libero accesso* a una classe derivata, indipendentemente dal `package` in cui essa è definita

Occorre dunque *cambiare la protezione del campo `val` nella classe `Counter`*

Ereditarietà - 9

## ESEMPIO

### Il contatore “riadattato”...

```
public class Counter {  
    protected int val;  
    public Counter()           { val = 1; }  
    public Counter(int v)     { val = v; }  
    public void reset()       { val = 0; }  
    public void inc()          { val++; }  
    public int getValue()     { return val; }  
}
```

Nuovo tipo di protezione!

Ereditarietà - 10

## ESEMPIO

... e il contatore con decremento:

```
public class Counter2
    extends Counter {
    public void dec() { val--; }
}
```

*Ora funziona !*

Ereditarietà - 11

## UNA RIFLESSIONE

La qualifica `protected`:

- rende accessibile un campo *a tutte le sottoclassi, presenti e future*
- costituisce perciò un *permesso di accesso "indiscriminato"*, valido per *ogni possibile sottoclasse che possa in futuro essere definita, senza possibilità di distinzione*

Ereditarietà - 12

# EREDITARIETÀ

## Cosa si eredita?

- **tutti i dati della classe base**
  - anche quelli privati, a cui comunque la classe derivata non potrà accedere direttamente
- **tutti i metodi...**
  - anche quelli che la classe derivata non potrà usare direttamente
- **... *tranne i costruttori*, perché sono specifici di quella particolare classe**

Ereditarietà - 13

## EREDITARIETÀ E COSTRUTTORI

- **Una classe derivata *non può prescindere dalla classe base*, perché ogni istanza della classe derivata *comprende in sé*, indirettamente, un oggetto della classe base**
- **Quindi, *ogni costruttore della classe derivata deve invocare un costruttore della classe base affinché esso costruisca la “parte di oggetto” relativa alla classe base stessa:***

***“ognuno deve costruire quello che gli compete”***

Ereditarietà - 14

# EREDITARIETÀ E COSTRUTTORI

- Perché bisogna che ogni costruttore della classe derivata invochi un costruttore della classe base?
  - solo il costruttore della classe base può sapere come inizializzare i dati ereditati in modo corretto
  - solo il costruttore della classe base può garantire l'inizializzazione dei dati privati, a cui la classe derivata non potrebbe accedere direttamente
  - è inutile duplicare nella sottoclasse tutto il codice necessario per inizializzare i dati ereditati, che è già stato scritto

Ereditarietà - 15

# EREDITARIETÀ E COSTRUTTORI

- *Ma come può un costruttore della classe derivata invocare un costruttore della classe base?*  
I costruttori non si possono mai chiamare direttamente!
- Occorre un modo per dire al costruttore della classe derivata di “*appoggiarsi*” a un opportuno costruttore della classe base: per questo si usa la parola chiave

**super**

Ereditarietà - 16

# ESEMPIO

Il contatore con decremento:

```
public class Counter2
```

Costruttore di default *generato automaticamente dal sistema* in assenza di altri costruttori

```
public void dec() { v--; }  
public Counter2() { super(); }  
public Counter2(int v) { super(v); }  
}
```

L'espressione `super(...)` invoca il costruttore della classe base che corrisponde come numero e tipo di parametri alla lista data

Ereditarietà - 17

## EREDITARIETÀ E COSTRUTTORI

- *E se non indichiamo alcuna chiamata a `super(...)`?*
- Il sistema inserisce automaticamente una chiamata al *costruttore di default* della classe base aggiungendo la chiamata a `super()`
- In questo caso il costruttore di default della classe base deve esistere, altrimenti si ha **ERRORE**

Ereditarietà - 18

# EREDITARIETÀ E COSTRUTTORI

**RICORDA:** il sistema genera automaticamente il costruttore di default solo se noi non definiamo alcun costruttore!

Se c'è anche solo una definizione di costruttore data da noi, il sistema assume che noi sappiamo il fatto nostro, e non genera più il costruttore di default automatico

- In questo caso il costruttore di default della classe base deve esistere, altrimenti si ha **ERRORE**

Ereditarietà - 19

## super: RIASSUNTO

La parola chiave *super*

- nella forma `super (...)`, invoca un costruttore della classe base
- nella forma `super.val`, consente di accedere al campo `val` della classe base (*sempre che esso non sia private*)
- nella forma `super.metodo()`, consente di invocare il metodo `metodo()` della classe base (*sempre che esso non sia private*)

Ereditarietà - 20

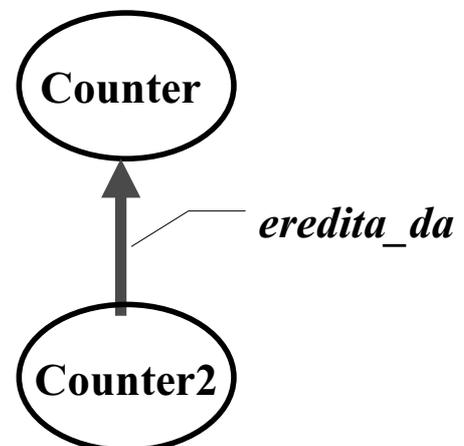
# COSTRUTTORI e PROTEZIONE

- Di norma, i costruttori sono `public`
  - in particolare, è sempre pubblico il costruttore di default generato automaticamente da java
- Almeno un costruttore pubblico deve sempre esistere, a meno che si voglia *impedire espressamente* di creare oggetti di tale classe agli utenti “non autorizzati”
  - caso tipico: una classe che fornisce solo costruttori protetti è pensata per fungere da classe base per altre classi più specifiche
  - non si vuole che vengano istanziati oggetti

Ereditarietà - 21

# EREDITARIETÀ: CONSEGUENZE

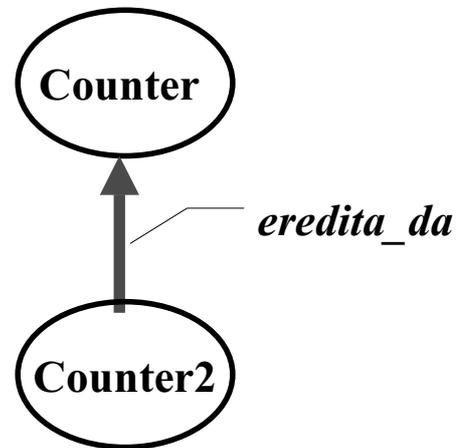
- Se una classe eredita da un'altra, *la classe derivata mantiene l'interfaccia di accesso della classe base*
  - anche se, naturalmente, può *specializzarla*, aggiungendo nuovi metodi
- Quindi, un `Counter2` può essere usato *al posto di un `Counter` se necessario*
- *Ogni `Counter2` è anche un `Counter` !*



Ereditarietà - 22

# EREDITARIETÀ: CONSEGUENZE

- Ogni oggetto di classe Counter2 è anche implicitamente di classe Counter
- *ma non viceversa*
  - un Counter è *meno ricco* di un Counter2
- Quindi, l'ereditarietà è più di un semplice “riuso di codice”: riusa l'astrazione
- *Induce una classificazione del mondo (aderente alla realtà...?)*



Ereditarietà - 23

# EREDITARIETÀ: CONSEGUENZE

- Se ogni Counter2 è anche un Counter, è possibile usare un Counter2 al posto di un Counter senza che il sistema se ne accorga!

```
public class Esempio6 {
    public static void main(String args[]) {
        Counter c1 = new Counter(10);
        Counter2 c2 = new Counter2(20);
        c2.dec(); // OK: c2 è un Counter2
        // c1.dec(); // NO: c1 è solo un Counter
        c1=c2; // OK: c2 è anche un Counter
        // c2=c1; // NO: c1 è solo un Counter
    }
}
```

Ereditarietà - 24

## EREDITARIETÀ: CONSEGUENZE

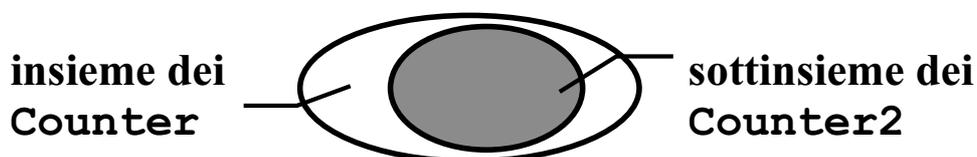
- Se ogni Counter2 è anche un Counter, è possibile usare un Counter2 al posto di un Counter senza che il sistema se ne accorga!

```
public class Esempio6 {  
    Counter c1; Counter2 c2; Counter2[] s[]) {  
        // OK perché c2 è un Counter2, quindi anche implicitamente un Counter  
        // (e c1 è a sua volta un Counter)  
        c2.decre(); // OK: c2 è un Counter2  
        // c1.decre(); // NO: c1 è solo un Counter  
        c1=c2;  
        // c2=c1;  
    }  
}
```

NO, perché c2 è un Counter2, e come tale esige un suo "pari", mentre c1 è "solo" un Counter

## EREDITARIETÀ: CONSEGUENZE

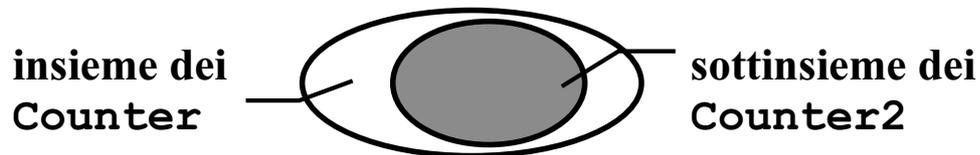
- Dunque, la classe Counter2 definisce un sottotipo della classe Counter



- Gli oggetti di classe Counter sono compatibili con gli oggetti di classe Counter2 (perché la classe Counter2 è inclusa nella classe Counter)
- ma non viceversa
- Ovunque si possa usare un Counter, lì si può usare un Counter2 (ma non viceversa)

# EREDITARIETÀ: CONSEGUENZE

- Dire che *ogni Counter2 è anche un Counter* significa dire che *l'insieme dei Counter2 è un sottoinsieme dell'insieme dei Counter!*

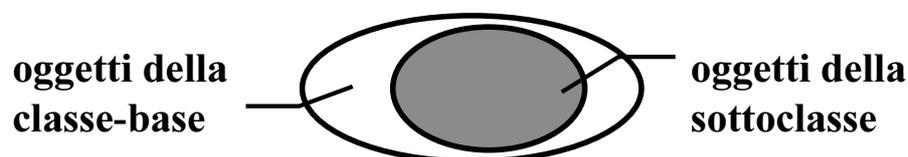


- Se questo è vero nella realtà, la classificazione è *aderente alla realtà del mondo*
- Se invece è falso, questa classificazione *nega la realtà del mondo*
  - non è un buon modello del mondo
  - può produrre assurdità e inconsistenze

Ereditarietà - 27

# EREDITARIETÀ: CONSEGUENZE

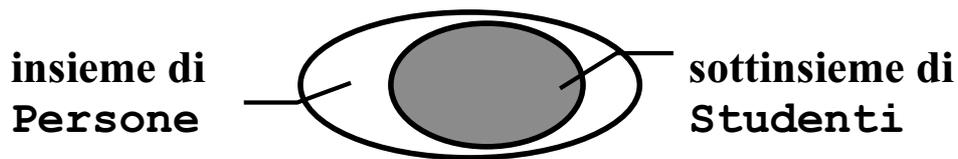
- Una sottoclasse dovrebbe dunque *delimitare un sottoinsieme* della classe da cui deriva: *altrimenti, si modella la realtà al contrario!*



- **Esempi**
  - Studente che deriva da Persona → OK (ogni Studente è *anche* una Persona)
  - Reale che deriva da Intero → NO (non è vero che ogni Reale *sia anche* un Intero!)

Ereditarietà - 28

# UN ESEMPIO COMPLETO



- Una classe `Persona`
- e una sottoclasse `Studente`
  - è aderente alla realtà, perché è vero nel mondo reale che tutti gli studenti sono persone
  - compatibilità di tipo: potremo usare uno studente (che è *anche* una persona) ovunque sia richiesta una generica persona  
ma non viceversa: se serve uno studente, non si può accontentarsi di una generica persona!

Ereditarietà - 29

## LA CLASSE `Persona`

```
public class Persona {
    protected String nome;
    protected int anni;
    public Persona() {
        nome = "sconosciuto"; anni = 0; }
    public Persona(String n) {
        nome = n; anni = 0; }
    public Persona(String n, int a) {
        nome=n; anni=a; }
    public void print() {
        System.out.print("Mi chiamo " + nome);
        System.out.println(" e ho " + anni + "anni");
    }
}
```

Ereditarietà - 30

## LA CLASSE Studente

```
public class Studente extends Persona {
    protected int matr;
    public Studente() {
        super(); matr = 9999; }
    public Studente(String n) {
        super(n); matr = 8888; }
    public Studente(String n, int a) {
        super(n,a); matr=7777; }
    public Studente(String n, int a, int m) {
        super(n,a); matr=m; }
    public void print() {
        super.print();
        System.out.println("Matricola = " + matr);
    }
}
```

Ereditarietà - 31

## LA CLASSE Studente

```
public class Studente extends Persona {
    protected int matr;
    public Studente() {
        super(); matr = 9999; }
    public Studente(String n) {
        super(n); matr = 8888; }
    public Studente(String n, int a) {
        super(n,a); matr=7777; }
    public Studente(String n, int a, int m) {
        super(n,a); matr=m; }
    public void print() {
        super.print();
        System.out.println("Matricola = " + matr);
    }
}
```

Ridefinisce il metodo void print()

- sovrascrive quello ereditato da Persona
- è una versione specializzata per Studente che però riusa quello di Persona (super), estendendolo per stampare la matricola

Ereditarietà - 32

## LA CLASSE EsempioDiCittà

```
public class EsempioDiCittà {  
    public static void main(String args[]){  
        Persona p = new Persona("John");  
        Studente s = new Studente("Tom");  
        p.print(); // stampa nome ed età  
        s.print(); // stampa nome, età, matricola  
        p=s; // OK (Studente estende Persona)  
        p.print(); // COSA STAMPA ???  
    }  
}
```

L'assegnamento `p=s` non comporta perdita di informazione, perché si assegnano riferimenti (gli oggetti puntati rimangono inalterati)

L'editore - 33

## LA CLASSE EsempioDiCittà

```
public class EsempioDiCittà {  
    p.print();  
    s.print(); // stampa nome, età, matricola  
    p=s;  
    p.print();  
}
```

- Se prevale la natura del riferimento, stamperà solo nome ed età
- Se prevale la natura dell'oggetto puntato, stamperà nome, età e *matricola*

**È un problema di POLIMORFISMO**

PROBLEMA: cosa stampa?

- `p` è un riferimento a `Persona`
- *ma gli è stato assegnato un oggetto `Studente`*

# POLIMORFISMO

- Un metodo si dice *polimorfo* quando è in grado di *adattare il suo comportamento allo specifico oggetto* su cui deve operare
- In Java, la possibilità di usare riferimenti a una data classe
  - ad esempio, `Persona`per puntare a oggetti *di classi più specifiche*
  - ad esempio, `Studente`introduce *in astratto* la possibilità di avere polimorfismo

*In pratica?*

Ereditarietà - 35

# POLIMORFISMO

In pratica, *dipende cosa prevale:*

- se prevale il tipo del riferimento, non ci sarà mai polimorfismo
  - in tal caso, `p.print()` stamperà solo nome ed età, perché verrà invocato il metodo `print()` della classe `Persona`
- se invece prevale il tipo dell'oggetto, allora c'è polimorfismo
  - in tal caso, `p.print()` stamperà nome, età e matricola, perché verrà invocato il metodo `print()` della classe `Studente`

Ereditarietà - 36

# POLIMORFISMO

In pratica, *dipende cosa prevale:*

- se prevale il tipo del riferimento  
Java supporta il Polimorfismo  
→ **prevale il tipo dell'oggetto** nome ed età, p... invoca il metodo print () della classe Persona

- se invece prevale il tipo dell'oggetto  
a...  
**LATE BINDING:** le chiamate ai metodi sono collegate alla versione opportuna del metodo al momento della chiamata, in base all'oggetto effettivamente referenziato (a "run-time")

Ereditarietà - 37

## LA CLASSE EsempioDiCittà

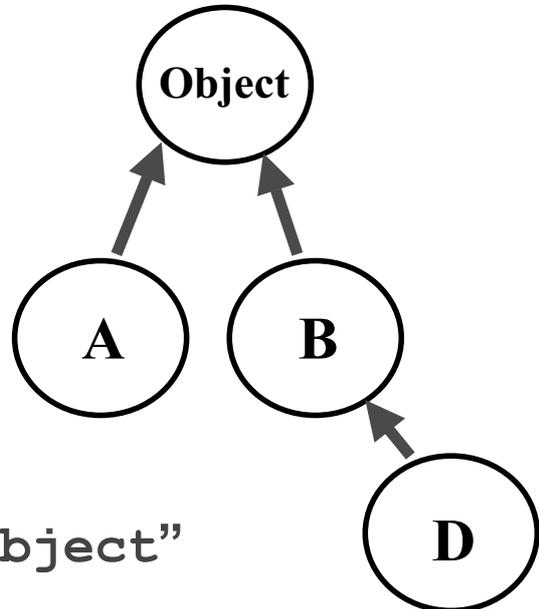
```
public class EsempioDiCittà {  
    public static void main(String args[]){  
        Persona p = new Persona("John");  
        Studente s = new Studente("Tom");  
        p.print(); // stampa nome ed età  
        s.print(); // stampa nome, età, matricola  
        p=s;  
        p.print(); // COSA STAMPA ???  
    }  
}
```

**void print()** è un metodo polimorfo  
Poiché p riferenzia uno Studente,  
stampa nome, età e matricola

Ereditarietà - 38

# GERARCHIE DI EREDITARIETÀ

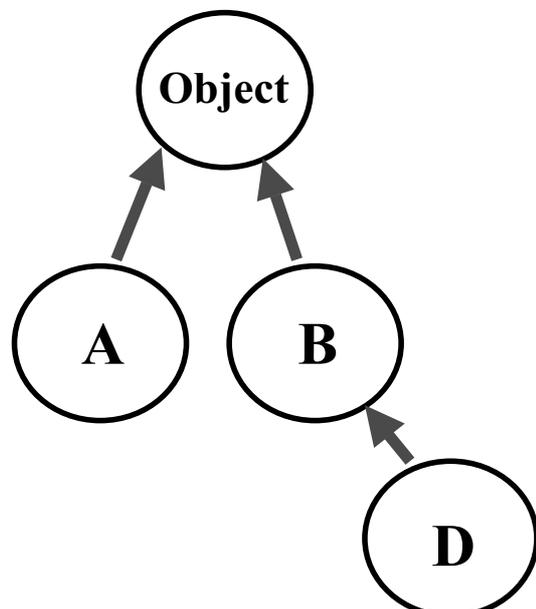
- La relazione di ereditarietà determina la nascita di *gerarchie* o *tassonomie* di ereditarietà
- In Java, *ogni classe deriva implicitamente dalla classe-base Object*, che è la radice della gerarchia
- La frase “`class A`” sottintende “`extends Object`”



Ereditarietà - 39

## Object, LA RADICE DI TUTTO

- La classe base *Object* definisce alcuni metodi, *ereditati da tutte le altre classi*:
  - `clone()`
  - `equals()`
  - `toString()`
  - ...



Ereditarietà - 40

# Object, LA RADICE DI TUTTO

## Alcuni metodi interessanti:

- `protected Object clone()`
  - duplica l'oggetto su cui è invocato
  - è un metodo protetto: per rendere disponibile questa funzionalità all'esterno di una classe, occorre *ridefinirlo come pubblico*
- `public boolean equals(Object x)`
  - definisce il criterio di uguaglianza fra oggetti
  - per default, è l'uguaglianza fra riferimenti
- `public String toString()`
  - crea una rappresentazione dell'oggetto sotto forma di stringa

Ereditarietà - 41

## ESEMPIO

Una piccola classe (*eredita implicitamente il metodo `toString()` da `Object`*):

```
public class Deposito {  
    float soldi;  
    public Deposito() { soldi=0; }  
    public Deposito(float s) { soldi=s; }  
}
```

Ereditarietà - 42

## ESEMPIO

... e una classe che la usa:

```
public class Esempio7 {  
    public static void main(String args[]) {  
        Deposito d1 = new Deposito(312);  
        System.out.println(d1);  
    }  
}
```

Per stampare d1, viene invocato automaticamente il metodo toString()

È una forma compatta per

System.out.println(d1.toString());

Ereditarietà - 43

## ESEMPIO

Se il toString() predefinito da Object non soddisfa, si può ridefinirlo:

```
public class Deposito {  
    float soldi;  
    public Deposito() { soldi=0; }  
    public Deposito(float s) { soldi=s; }  
    public String toString() {  
        return "Deposito di valore " + soldi;  
    }  
}
```

Viene creato un nuovo oggetto String concatenando la frase "Deposito di valore " con il risultato di Float.toString(soldi)

# ESEMPIO

L'output nel primo caso...

```
Deposito@712c1a3c
```

Identificativo univoco generato da Java:  
nome della classe + indirizzo dell'oggetto

... e nel secondo caso:

```
Deposito di valore 312.0
```

Ereditarietà - 45

## CLASSI FINALI

- Una classe finale (`final`) è una classe di cui si vuole *impedire a priori* che possano essere definite, un domani, delle sottoclassi
- Esempio:

```
public final class TheLastCounter  
    extends Counter {  
    ...  
}
```

Ereditarietà - 46