

La sottoclasse chiama un costruttore della superclasse per inizializzare i campi ereditati da quest'ultima.

Per chiamare il costruttore della superclasse si usa super.

La chiamata è super (...) dove essere la prima istruzione del costruttore della sottoclasse.

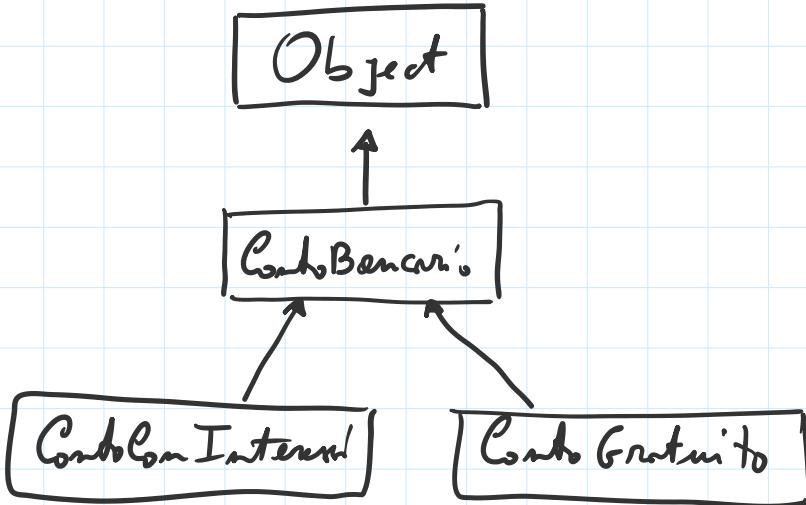
Se la superclasse è dotata di un costruttore privo di argomenti allora la chiamata è super() può essere omessa.

Se la superclasse è dotata di soli costruttori con argomenti allora è obbligatorio chiamarne uno con super (...).

Esiste una classe di sistema che si chiama Object (package java.lang)

Se una classe non dichiara esplicitamente di volersi estendere un'altra allora è implicitamente figlia di Object.

La gerarchia reale è:



La classe `Object` è dotata di alcuni metodi, che vengono quindi ereditati da tutte le classi Java.

I metodi di `Object` possono essere divisi in due categorie

- metodi per la sincronizzazione fra thread
(`wait()`, `notify()`, `notifyAll()` che vedremo più avanti)
- metodi di utilità generale
(`equals()`, `hashCode()`, `toString()`)

Il metodo `equals()`

`public boolean equals (Object o)`

Confronta l'oggetto implicito con l'oggetto `o` e restituisce `true` se i due oggetti sono uguali, `false` altrimenti

Ogni classe può ridefinire il metodo equals secondo il proprio criterio.

Per esempio: la classe String ridefinisce il metodo in modo tale da verificare se le due stringhe sono composte dagli stessi caratteri.

L'implementazione della classe Object del metodo equals() corrisponde a `a == b` (i riferimenti puntano allo stesso oggetto).

Se definisco A senza ridefinire equals()

A a1 = new A();

A a2 = new A();

boolean r = a1.equals(a2);

// è come scrivere

boolean r = a1 == a2;

il risultato (r) è falso perché sono due oggetti distinti

Per la classe Punto potremo ridefinire il metodo equals() così:

```
public class Punto {  
    private double x;  
    private double y;  
    :  
    public boolean equals( Object o ) {  
        mi accorgo  
        che non sta  
        un riferimento  
        nullo  
        (altrimenti  
        restituisce false)  
        if ( o == null ||  
            getClass() != o.getClass() ) {  
            return false  
        }  
        Punto p = (Punto) o;  
        return x == p.x && y == p.y;  
    }  
    :  
}
```

tipo Object
(non Punto)

verifica se
la classe di
o no la
stessa
classe
dell'
oggetto
implementato

cost
convertito
a Punto

Nota: il metodo getClass() restituisce la classe dell'oggetto su cui viene applicato

Nella classe Studenta il metodo equals() potrebbe essere ridefinito come

```
public boolean equals(Object o) {
```

```
    if (o == null ||
```

```
        o.getClass() != getClass())
```

```
        return false;
```

```
}
```

```
Studente s = (Studente) o;
```

```
return matrcola == s.matrcola;
```

```
}
```

```
↑
```

due studenti sono uguali
se hanno la stessa matrcola

Il metodo hashCode

```
public int hashCode()
```

restituisce l'hashcode dell'oggetto su cui viene applicato

L'implementazione predefinita della classe Object restituisce valori diversi per oggetti diversi (tipicamente l'indirizzo interno alla JVR, a partire dal quale l'oggetto è allocato)

Molti classi "contenitore" del linguaggio "pretendono" che se il metodo equals restituisce true per oggetti diversi (perché ridefinito) allora il metodo hashCode deve restituire lo stesso valore per tali ...

dove restituire lo stesso valore per tutti gli oggetti.

Nella classe Studente potremmo ridefinirlo così:

```
public int hashCode() {  
    return matricola;  
}
```

Il metodo `toString()`

```
public String toString()
```

restituisce una rappresentazione in formato stringa dell'oggetto

l'implementazione predefinita di Object restituisce una stringa

NomeDellaClasse@hashCodeDellOggetto

il metodo `toString()` viene richiesto automaticamente quando concateniamo un oggetto a una stringa

```
Studente s1 = new Studente("Masi", "Rossi");
```

String x = "Studente: " + s1;

è come scrivere

String x = "Studente: " + ss.toString();

x vale "Studente: Studente@A39x.."

possiamo ridefinire toString() così:

```
public String toString() {
    return nome + ", " + cognome;
}
```

e otteniamo

x vale "Studente: Mario, Rossi"

Anche il metodo finalize() lo ereditiamo
dalla classe Object.

Riferimenti e ereditarietà

Un riferimento di tipo superclasse può puntare
a oggetti del tipo sottoclasse

ContoBancario cb;

ContoConInteressi cci = new ContoConInteressi("Mario", 0.01);

c b = cci; // OK: un ContoConInteressi è un
// ContoBancario

ContoBancario cb1 = new ContoConInteressi("Luigi", 0.02);

Object o1 = new ContoConInteressi("Sera", 0.03);

E' possibile convertire da supertipo a sottotipo. La conversione deve essere esplicita con un cast (Tipo a cui voglio convertire)

Viene eseguito un controllo e se l'oggetto non può essere convertito nel tipo specificato allora viene lanciata una ClassCastException (segnalazione d'errore)

ContoConInteressi x = new ContoConInteressi("Peach", 0.04);

ContoBancario y = x;

ContoConInteressi z;

z = (ContoConInteressi) y;

↑ questa conversione deve essere esplicita
OK: il tipo nelle due classi è lo stesso

esplicitamente

OK: il tipo reale dell'oggetto puntato
da y è ContoConInteressi

ContoBancario a = new ContoBancario("Toad");

ContoConInteressi b;

b = (ContoConInteressi) a;

T genera ClassCastException: il tipo dell'
oggetto non è "compatibile" con ContoConInteressi

Quello che conta è il tipo reale
dell'oggetto (non del riferimento)

Object o = new ContoConInteressi("Mars", 0.01);

ContoBancario i = (ContoBancario) o; //OK: un ContoConInt.
è un ContoBancario

ContoConInteressi j = (ContoConInteressi) o; //OK

Riferimenti di tipo diversi che puntano allo stesso oggetto

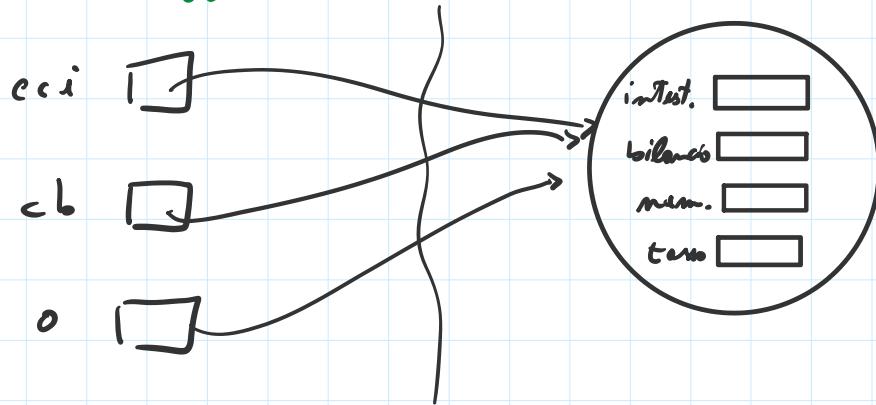
ContoConInteressi ee i = new ContoConInteressi("Luigi", 0.02);

ContoBancario cb = ee i;

Object o = ee i;

Tre riferimenti di tipo diversi che puntano
allo stesso oggetto

che riguardano un tipo diverso che puntano
allo stesso oggetto



Il tipo del riferimento determina
l'esecuzione dei metodi che possono essere
invocati

Il metodo che viene effettivamente invocato
dipende dal tipo dell'oggetto (non del riferimento)

Nel caso precedente:

cci. aggiungiInteressi(); // OK
cci. deposita(100.0); // OK : ereditato da ContoBancario
cci. preleva(50.0); // OK : ereditato da ContoBancario
int k = cci. hashCode(); // OK : ereditato da Object

cb. aggiungiInteressi(); // errore
cb. deposita(30.0); // OK
cb. preleva(20.0); // OK
k = cb. hashCode(); // OK

o. aggiungiInteressi(); // errore
o. deposita(20.0); // errore
o. preleva(10.0); // errore
k = o. hashCode(); // OK

Quando scrivo

String s = o.toString();

viene eseguito il `toString()` di
`ContoConInteressi`!

Altro esempio:

ContoBancario x = new ContoBancario("Maro.");

ContoConInteressi y = new ContoConInteressi("Enrico", 0.01);
x.transferring(y, 100.0);

Il metodo `transferring`, che è stato pensato per `ContoBancario`, funziona anche con `ContoConInteressi` (OK, un `ContoConInteressi` è un `ContoBancario`)

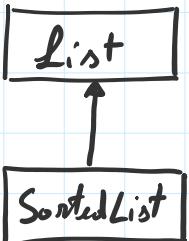
L'operatore `instanceof` può essere utile per capire quale sia il tipo di un oggetto per esempio prima di eseguire un cast

if (r instanceof ContoConInteressi) {

ContoConInteressi s = (ContoConInteressi) r;

restituisce true se r è un
o è un + -

restituisce true se r è un
ContoConInteressi:



```

void sort(List l) {
    if (l instanceof SortedList)
        return;
    // sort the list
}
  
```

Vediamo la classe ContoGratuito:

un certo numero di operazioni sono gratuite, poi si comincia a pagare

```

public class ContoGratuito extends ContoBancario {
    private static final double COSTO_OPERAZIONE = 1.0;
    private final int operazioniGratuite;
    private int operazioniEseguite;

    public ContoGratuito(String i, int n) {
        super(i);
        operazioniGratuite = n;
    }

    public int operazioniGratuiteRimanenti(){
        return operazioniGratuite - operazioniEseguite;
    }

    public void deposita(double d) {
        operazioniEseguite++;
        super.deposita(d);
    }

    public void preleva(double d) {
        operazioniEseguite++;
        super.preleva(d);
    }

    public void assegnaCosti() {
        if(operazioniEseguite > operazioniGratuite) {
            double importo = (operazioniEseguite -
  
```

m° operazioni fatte

m° operazioni gratuite

Costo delle operazioni aggiuntive

chiama costruttore di ContoBancario per intestatario

nuovo metodo

Ridefinizione di deposita() e preleva()

richiede il deposito() di ContoBancario.

Dovrò mettere super. altrettanto

super.
 altri metodi
 richiamerai
 questo
 stesso
 metodo
 da punto

```

public void assegnaCosti() {
    if(operazioniEseguite > operazioniGratuite) {
        double importo = (operazioniEseguite -
        operazioniGratuite)*COSTO_OPERAZIONE;
        // richiamo preleva della superclasse in modo
        // che non conti come operazione
        super.preleva(importo);
    }
    operazioniEseguite = 0;
}

public String toString(){
    String s = super.toString();
    return s + ", op. gratuite: " +
    operazioniGratuiteRimanenti();
}
}

```

Richiamo `toString()`
 da `ContoBancario`.

Se avessi scritto `deposita()` così:

```

public void deposita (double d) {
    operazioniEseguite++;
    deposita (d);
}

```

NO: ricorsione infinita

super e this

this

- è un riferimento all'oggetto implementante
- posso usarlo come prima istruzione
di un costruttore per richiamare
un altro costruttore della stessa classe

super

- non è un riferimento
- posso usarlo come prima istruzione

- posso usarlo come prima istruzione di un costruttore per richiamare un costruttore della superclasse
- permette di invocare un metodo della superclasse che è stato ridefinito nella classe corrente

Polidormorfismo

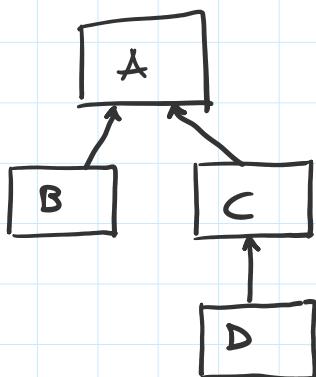
Un'entità del nostro programma può avere diverse forme

Supponiamo che delle sottoclassi ridefiniscono un metodo della superclasse

Un riferimento di tipo superclasse può puntare a istanze delle sottoclassi

Quando viene invocato il metodo , sul riferimento di tipo superclasse, l'esecuzione si diverte e dipende dalla ridefinizione fatta dalla sottoclasse a cui l'oggetto appartiene.

Esempio



```
class A {  
    void m() {  
        System.out.println("m di A");  
    }  
}
```

```
class B extends A {  
    void m() {  
        System.out.println("m di B");  
    }  
}
```

```
class C extends A {  
    void m() {  
        System.out.println("m di C");  
    }  
}
```

```
class D extends C {  
    void m() {  
        System.out.println("m di D");  
    }  
}
```

A r1 = new A();

A r2 = new B();

A r3 = new C();

A r₄ = new D();

R1. m(); // Stampa m di A

R2. m(); // Stampa m di B

R3. m(); // Stampa m di C

R4. m(); // Stampa m di D

 L'uso sempre lo stesso nome per il metodo, vengono eseguite versioni diverse (in diverse forme)

A r₅;

D r₆ = new D();

r₅ = r₆;

r₅.m(); // Stampa m di D

ContoBancario x = new ContoBancario("Mario");

ContoGratuito y = new ContoGratuito("Luigi", 100);

x.transfer(y, 200.0);

 posso usare ContoGratuito in tutto il codice scritto per ContoBancario

usa l'implementazione di deposita() di ContoGratuito

PA2021_ereditarietà

The screenshot shows an IDE interface with three main panes: Files, Main.java, and Console.

Files pane:

- Main.java
- ContoBancario.java
- ContoConInteress...
- ContoGratuito.java

Main.java code:

```
1 class Main {  
2     public static void main(String[] args) {  
3         ContoConInteressi cci = new ContoConInteressi("Mario",  
4             0.02);  
5         ContoGratuito cg = new ContoGratuito("Furio", 10);  
6         System.out.println(cci);  
7         System.out.println(cg);  
8  
9         cci.deposita(200.0);  
10        cg.deposita(300.0);  
11        cci.trasferisci(cg, 50.0);  
12        System.out.println(cci);  
13        System.out.println(cg);  
14  
15        ContoBancario cb1;  
16        cb1 = cg;  
17        cb1.preleva(20.0);  
18        System.out.println(cg);  
19    }  
20}
```

Console pane:

```
> javac -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest-core-1.3.jar:/run_dir/json-simple-1.1.1.jar -d . ContoBancario.java ContoConInteressi.java ContoGratuito.java Main.java  
> java -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest-core-1.3.jar:/run_dir/json-simple-1.1.1.jar Main  
intestatario: Mario, numero: 1, bilancio: 0.0, tasso: 0.02  
intestatario: Furio, numero: 2, bilancio: 0.0, op. gratuita: 10  
intestatario: Mario, numero: 1, bilancio: 150.0, tasso: 0.02  
intestatario: Furio, numero: 2, bilancio: 350.0, op. gratuita: 8  
intestatario: Furio, numero: 2, bilancio: 330.0, op. gratuita: 7  
> []
```