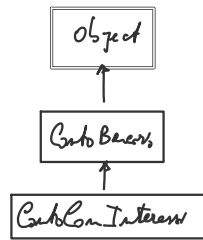
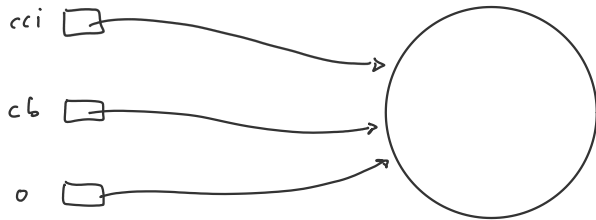


Riferimenti e oggetti



```

ContoConInteressi cci = new ContoConInteressi("Gino", 0.1);
ContoBancario cb = cci; // ok ContoBancario è un supertipo
Object o = cci; // ok Object è un supertipo
  
```



```

cci.aggiungiInteressi(); // ok: è metodo di ContoConInteressi
cci.deposito(100.0); // ok: è un metodo ereditato da ContoBancario

cb.aggiungiInteressi(); // Errore: è un riferimento di tipo ContoBancario; i metodi che è possibile invocare sono quelli di ContoBancario + quelli ereditati
cb.deposito(100.0); // OK: deposito è di ContoBancario

o.aggiungiInteressi(); // Errore
o.deposito(100.0); // Errore
  
```

Posso invocare solo i metodi della classe Object

Perdi assegnare un oggetto a un riferimento che appartiene a un suo supertipo?

Risposta del Codice: tutto il codice scritto per il supertipo può funzionare con il sottotipo

```

ContoBancario x = new ContoBancario("Pippo");
ContoConInteressi y = new ContoConInteressi("Plo", 0.1);
x.transferisci(y, 10.0);
  
```

il codice di transferisci era pensato per ContoBancario, ma funzionerà anche su ContoConInteressi

```

class Rapporto {
    public void produci(ContoBancario c) {
    }
}
  
```

scritto per ContoBancario

```

Rapporto r = new Rapporto(-);
r.produci(y);
  
```


Conto Con Interessi:

- Conoscere il tipo reale di un oggetto

```

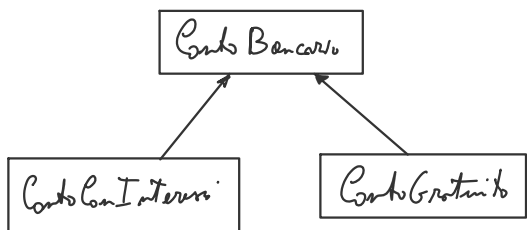
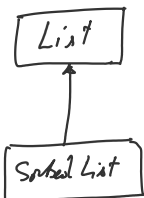
if ( r instanceof ColorInterim ) {

```

}  que potred fore un
Cant :

$$C_{\text{atolun Interem}} c = (C_{\text{atolun Interem}})' r;$$

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



```

public class ContoCorrente extends ContoBancario {
    private static final double costo = 1.5;
    private final int opCorrente;
    private int opEseguite;

    public ContoCorrente(String i, int m) {
        super(i);
        opCorrente = m;
    }

    public int opCorrenteRimanenti() {
        return opCorrente - opEseguite;
    }

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

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

    public void assegnaCosti() {
        if (opCorrente < opEseguite) {
            double ii = (opEseguite - opCorrente) * costo;
            super.preleva(ii);
        }
        opCorrente = 0;
    }
}

```

Riduzione di deposito
 richiamo deposita della
 superclasse (senza super richiamerbbe
 se stesso).
 Come super
 richiamo preleva della
 superclasse ContoBancario
 se scrivo solo preleva
 richiamo quello della classe
 corrente (che aumenta
 inutilmente il contatore)

```

public String toString() {
    String s = super.toString();
    return s + ", op. gestante rimanenti: " +
        opGestanteRimanenti();
}
}

```

Riepilogo relativo all'uso di this e super

this

- è un riferimento all'oggetto implicito
- può essere usato come prima istruzione di un costruttore per richiamare un altro costruttore della stessa classe
- utile come risolutore di ambiguità

super

- non è un riferimento
- come prima istruzione di un costruttore può essere usato per richiamare un costruttore della superclasse
- consente di chiamare un metodo della superclasse che è stato ridefinito nella classe corrente

Una sottoclasse

- eredita i metodi della superclasse
- può definire nuovi metodi (che si vanno ad aggiungere a quelli ereditati)
- può ridefinire (overriding) il comportamento di metodi della superclasse

Una sottoclasse

- eredita le variabili della superclasse
- può definire nuove variabili (che si vanno ad aggiungere a quelle ereditate)
- può ridefinire (mascherare) variabili della superclasse definendone di proprie con lo stesso nome

In caso di ridefinizione di variabili il tipo del riferimento determina quale sia la variabile acceduta

Esempio

```
public class Sopra {
```

```

String variabile = "variabile di Sopra";

public void metodo() {
    System.out.println("Metodo di Sopra, " + variabile);
}

public class Sotto extends Sopra {

    String variabile = "variabile di Sotto";

    public void metodo(){
        System.out.println("Metodo di Sotto, " + variabile);
    }
}

```

nasconde variabile della superclass

qui accediamo a quella della classe corrente

```

class Main {
    public static void main(String[] args) {
        Sotto sub = new Sotto();
        Sopra up = sub;

        System.out.println(up.variabile);
        System.out.println(sub.variabile);

        up.metodo();
        sub.metodo();
    }
}

```

} → il tipo del riferimento determina quale delle due viene acceduto

il metodo eseguito è sempre quello della sottoclasse

[esempio riferimenti](#)

In generale non è utile ridefinire variabili della superclass.

È qualcosa che può succedere quando superclass e sottoclasse sono scritte da sviluppatori diversi (non coauthors).

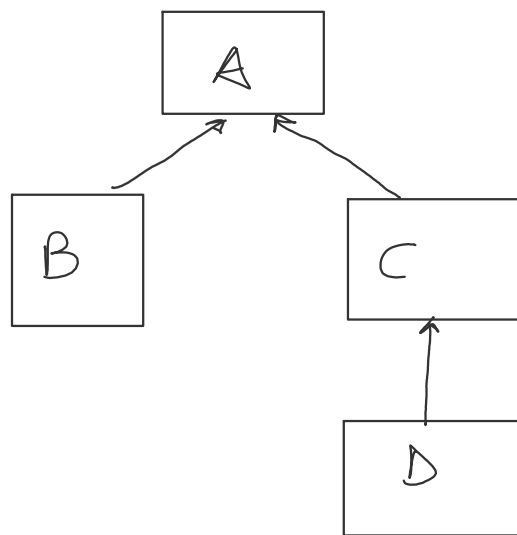
Polimorfismo

- Supponiamo che una sottoclasse ridefinisca (overriding) un metodo della superclass.
- Un oggetto sottoclasse può essere eseguito a un riferimento di tipo superclass.

- Quando usando un riferimento superclass viene invocato il metodo, il codice che viene eseguito è quello della sottoclasse.

Stesso nome ma operazioni "diverse"

Esempio



```

class A {
    void met() {
        System.out.println("Metodo di A");
    }
}

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

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

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

A r1 = new A();

A r2 = new B();

A r3 = new C();

A r4 = new D();

```


r1. met(); // esegue met() di A
r2. met(); // esegue met() di B
r3. met(); // esegue met() di C
r4. met(); // esegue met() di D

```

```

ContoBancaio cb = new ContoBancaio("Pino");
ContoGratuito cg = new ContoGratuito("Gino", 100);
cb.transferisci(cg, 50.0);

```


 Per via del polimorfismo la chiamata
 a deposito fatta dentro transferisci
 manda in esecuzione il codice di deposito
 così come definito da ContoGratuito

```
String tmp = "Conti: " + cb + cg;
```

Classi Astratte

- Una classe astratta è una classe che non può essere istanziata
- È però possibile creare riferimenti di tipo classe astratta
- Una classe astratta può contenere del codice
 - in termini di variabili / costanti
 - in termini di metodi

Per definire una classe astratta:

```

abstract class OggettoGrafico {
    int x, y;
    void sposta(int newX, int newY) {
        // ...
    }
}

```

È possibile dichiarare metodi abstract

```
abstract void esempio(int x);
```

Se una classe ha almeno un metodo abstract

allora la classe deve essere abstract

Implementazione più semplice:

```
abstract class OggettoGrafico {  
    int x, y;  
    void sposta(int mx, int my) {  
        =  
    }  
    abstract void disegna();  
}
```

```
class Bottone extends OggettoGrafico {  
    void disegna() {  
        =  
        // disegna il bottone  
    }  
}
```

```
class Etichetta extends OggettoGrafico {  
    String testo;  
    void disegna() {  
        =  
        // Disegna l'etichetta  
    }  
}
```

```
OggettoGrafico o1 = new Bottone();  
OggettoGrafico o2 = new Etichetta();  
o1.sposta(10, 20);  
o1.disegna();  
o2.sposta(30, 40);  
o2.disegna();  
  
OggettoGrafico o3 = new OggettoGrafico(); // errore
```

Classi e metodi finali

Una classe può essere marcata final

```
final class A {  
    =  
}
```

Una classe in java non può essere estesa

In una classe `final` tutti i metodi sono `final` (non possono essere ridefiniti).

Una classe può essere marcata come `final` per motivi di "sicurezza", per evitare che sottotipi "Rompano" il contratto che quella classe ha con l'esterno.

È possibile indicare come `final` il singolo metodo

```
class X {  
    final boolean controllaPassword(String p) {  
        ==  
    }  
    ==  
}
```

Non è possibile ridefinire `controllaPassword`, per esempio per restituire sempre `true`.

Usare `final` con parsimonia, è un limite al riuso del codice

La chiamata a un metodo `final` può essere leggermente più veloce (la JVM può sostituire la chiamata con codice `in-line`).

Interface

- Le interfacce Java definiscono protocolli di comportamento
- Sono collezioni di metodi `abstract`
- Chi aderisce al protocollo di comportamento deve fornire un'implementazione dei metodi dell'interfaccia

```
public interface Comparabile {  
    int compara(Object o);  
}
```

File `Comparable.java`
bytecode in
`Comparable.class`

implicitamente `public` e `abstract`


```

public class Data implements Comparabile {
    int giorno, mese, anno;

    public int compara (Object o) {
        Data d = (Data) o;
        if (anno != d.anno)
            return anno - d.anno;
        if (mese != d.mese)
            return mese - d.mese;
        return giorno - d.giorno;
    }
}

```

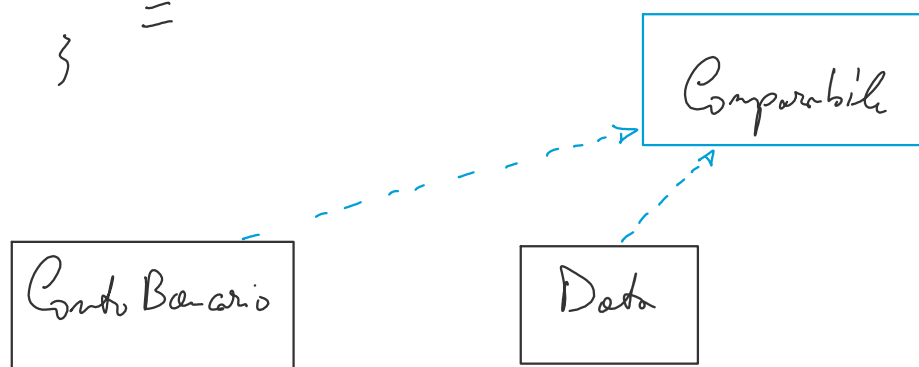
Supponiamo di voler fare lo stesso con per Conto Bancario

```

public class ContoBancario implements Comparabile {
    private double bilancio;

    =
    public int compara (Object o) {
        ContoBancario c = (ContoBancario) o;
        if (bilancio < c.bilancio) return -1;
        if (bilancio > c.bilancio) return +1;
        return 0;
    }
}
=
}

```



Una classe può implementare tutte le interfacce che vuole

```

class A implements Interf1, Interf2 {
    dobbiamo implementare tutti
    i metodi di Interf1
    e di Interf2
}

```

Non si possono creare oggetti di tipo
interfaccia

Posso creare riferimenti di tipo interfaccia
- che poi possono puntare a oggetti
che implementano quell'interfaccia

```
Comparable c1 = new Comparable(); // error
```

```
Comparable c2 = new Data();
```

```
Comparable c3 = new ContoBanca("Gino");
```

```
int r = c2.compara(new Data(-));
```

```
int s = c3.compara(new ContoBanca(-));
```