Capitolo 4:

Ereditarietà, dynamic binding e polimorfismo

Una proprietà importante delle classi di Java è legata alla possibilità di progettare una nuova classe per *estensione* di una classe esistente, dunque per *differenza*. Tutto ciò permette di concentrarsi solo sulle novità introdotte dalla nuova classe e di ereditare quelle che si mantengono inalterate, favorendo in tal modo la *produttività* del programmatore

Di seguito si considera una classe ContoBancario che definisce le usuali operazioni di deposito e prelievo. Un conto è identificato da un numero espresso mediante una String, e si caratterizza per il suo bilancio. Non è permesso al bilancio di andare "in rosso", ossia un prelevamento oltre il valore del bilancio non viene consentito. A questo scopo il metodo preleva() ritorna una booleana il cui valore true indica la conclusione con successo dell'operazione, il valore false il suo fallimento. Metodi accessori permettono di conoscere il numero di conto e il valore corrente del bilancio

Una classe ContoBancario

```
package poo.banca;
import java.io.*;
public class ContoBancario{
   private String numero;
   private double bilancio=0;
   public ContoBancario( String numero ){//primo costruttore
      this.numero=numero;
   }
   public ContoBancario( String numero, double bilancio ){//secondo costruttore
      this.numero=numero; this.bilancio=bilancio;
   }
   public void deposita( double quanto ){ //pre: quanto>0
      bilancio=bilancio+quanto; //o bilancio +=quanto;
   }//deposita
   public boolean preleva( double quanto ){ //pre: quanto>0
      if( quanto>bilancio ) return false;
      bilancio -= quanto;
      return true;
   }//preleva
   public double saldo(){ return bilancio; }
   public String conto(){ return numero; }//conto
   public String toString(){
      return String.format( "conto=%s bilancio=E %1.2f", numero, bilancio );
   }//toString
}//ContoBancario
```

In un main, ad es., si può quindi avere:

```
ContoBancario cb=new ContoBancario("51/554422",1000);
...
cb.deposita( 200 );
...
if( cb.preleva( 250 ) ) "corri alla posta a pagare la bolletta dell'ENEL"
...
System.out.println( cb );
```

Un conto bancario con fido

ContoBancario va bene per i clienti "ordinari". La banca dispone di un altro tipo di conto, ContoConFido, riservato a clientela selezionata, che ammette l'andata in rosso controllata da un fido. Chiaramente ContoConFido mantiene molte caratteristiche di ContoBancario ma in più introduce delle differenze (fido, bilancio in rosso etc.). Java consente di programmare una classe come ContoConFido per *specializzazione* (estensione o extends) della classe esistente ContoBancario:

Una classe ContoConFido erede di ContoBancario:

```
package poo.banca;
import java.io.*;
public class ContoConFido extends ContoBancario {
    private double fido=1000; //default
    public ContoConFido( String numero ){ super( numero ); }
    public ContoConFido( String numero, double bilancio ){
        super( numero, bilancio );
    }
    public ContoConFido( String numero, double bilancio, double fido ){
        super( numero, bilancio ); this.fido=fido;
    }
    public boolean preleva( double quanto ){
        //pre: quanto>0
        if( quanto<=saldo()+fido ){ super.preleva(quanto); return true; }
        return false;
    }//preleva
    public double fido(){ return this.fido; }
    public void nuovoFido( double fido ){ this.fido=fido; }
    public String toString(){
        return String.format( super.toString()+" fido=E %1.2f", fido );
    }
}//ContoConFido
```

Il pronome super

Si nota l'uso del pronome super per riferirsi alla super classe, ad esempio per invocare esplicitamente un costruttore della super classe cui si delega parte del processo di costruzione. Quando super è usato per questi scopi dev'essere la prima istruzione del costruttore. Si nota ancora che l'implementazione mostrata consente effettivamente che il bilancio possa diventare negativo, pur nei limiti del fido. Per altro, essendo private il campo bilancio di ContoBancario, ogni sua modifica va ottenuta mediante i metodi di ContoBancario.

Un'implementazione di ContoConFido con gestione dello "scoperto":

In questa versione della classe ContoConFido, il bilancio materialmente non diventa mai <0. L'andata in rosso è riflessa dal valore positivo di una cariabile di istanza scoperto. È necessario ridefinire non solo preleva() ma anche deposita() per mantenere aggiornato lo scoperto.

```
public class ContoConFido extends ContoBancario{
    private double fido=1000;
    private double scoperto=0;
    public ContoConFido( String numero ){
        super( numero );
    }
```

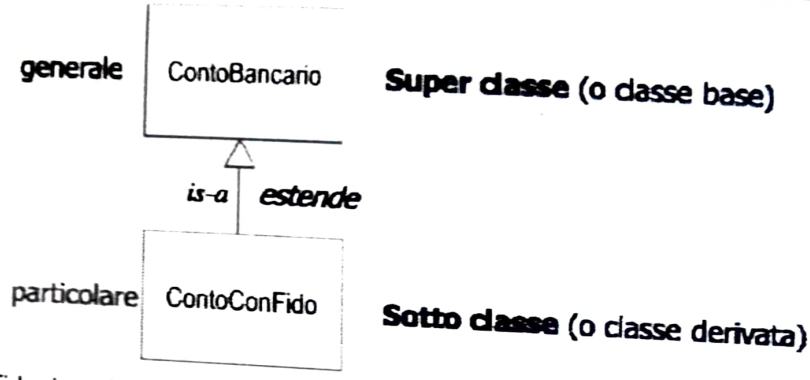
```
   public ContoConFido( String numero, double bilancio ){
      super( numero, bilancio );
   }
   public ContoConFido( String numero, double bilancio, double fido ){
      super( numero, bilancio ); this.fido=fido;
   }
   public void deposita( double quanto ){
      //pre: quanto>0
      if( quanto<=scoperto ){ scoperto-=quanto; return; } // copre lo scoperto
      double residuo=quanto-scoperto; // quanto rimane dopo aver coperto lo scoperto?
      scoperto=0;
      super.deposita( residuo );
   }//deposita
   public boolean preleva( double quanto ){
      //pre: quanto>0
      if( quanto<=saldo() ){ // posso i soldi sul conto senza sfruttare il fido
         super.preleva( quanto );
         return true;
      }
      if( quanto<=saldo()+fido-scoperto ){ // prelevo usando il fido
         double residuo=saldo();
         super.preleva( residuo ); // preleva tutto
         scoperto+=quanto-residuo;
         return true;
      }
      return false;
   }//preleva
   public double fido(){ return this.fido; }//fido
   public void nuovoFido( double fido ){
      this.fido=fido;
   }//nuovoFido
   public double scoperto(){
      return scoperto;
   }//scoperto
   public String toString(){
      return String.format( super.toString()+" fido=E %1.2f scoperto=E %1.2f", fido, scoperto );
   }//toString
}//ContoConFido
```

Seguono alcuni esempi d'uso:

```
ContoBancario c1=new ContoBancario("51/12345",2000);
ContoConFido c2=new ContoConFido("52/334455",10000,5000);
...
c1.deposita(20);
...
c2.deposita(2000);
...
c1.preleva( 240 );
c2.preleva( 13000 );
...
```

```
c1.fido(); //ERRORE: ContoBancario non conosce il concetto di fido
...
c2.nuovoFido( 8000 );
System.out.println( "nuovo fido="+c2.fido() );
```

Relazione di ereditarietà

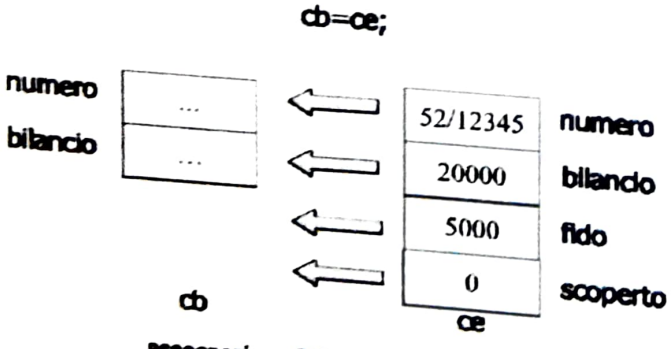


Si dice che ContoConFido è-*un* (*is-a*) ContoBancario, solo un pò più specializzato. ContoConFido è una sotto-classe (o classe derivata), ContoBancario una super-classe (o classe base). La relazione di ereditarietà da ContoConFido a ContoBancario è una relazione di generalizzazione (si veda anche il cap. 19).

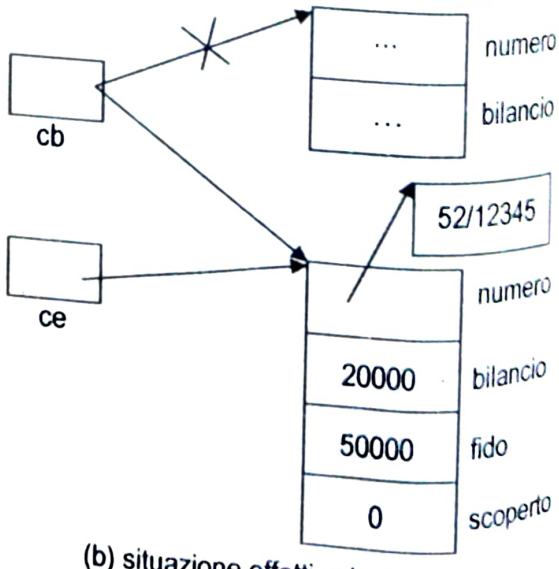
La relazione di ereditarietà è ben definita se un oggetto della classe derivata si può utilizzare in tutti i contesti in cui è atteso un oggetto della classe base (*principio di sostituibilità dei tipi*): In fondo: un conto confido è un conto bancario, solo un pò più particolare. Tuttavia: un conto bancario ***non*** **è** un conto con fido. Se un'applicazione richiede un conto con fido, non gli si può dare un conto bancario semplice! La parantela esistente tra classe base e classe derivata consente quanto segue:

```
ContoBancario cb=new ContoBancario(...);
...
ContoEsteso ce=new ContoEsteso(...);
...
cb=ce; //assegnazione dal particolare al generale OK
```

Assegnazione tra oggetti come "proiezione"



(a) situazione logica



(b) situazione effettiva in Java

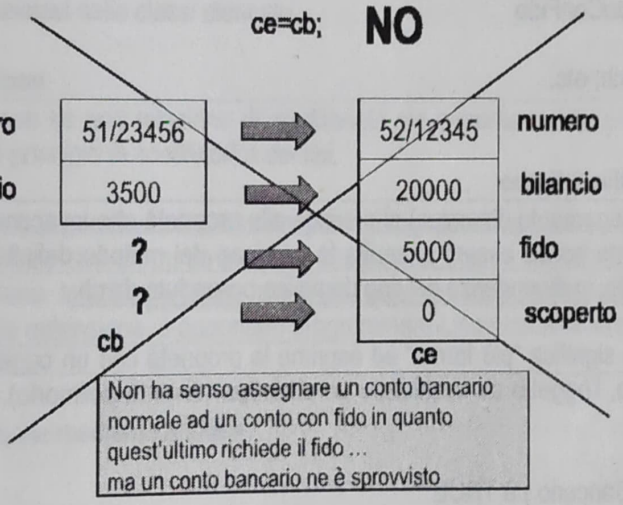
L'assegnazione da particolare a generale corrisponde, ad es., alla proiezione di un punto dello spazio cartesiano (con coordinate x, y e z) sul piano X-Y (la coordinata z è ignorata). Nella situazione effettiva di Java, a seguito dell'assegnazione cb=ce, cb punta all'oggetto composito riferito da ce. Tuttavia, cb lo vede con gli "occhiali" imposti dalla *sua* classe di appartenenza ContoBancario. Pertanto i campi fido e scoperto, anche se effettivamente presenti nell'oggetto puntato da cb, sono ignorati.

Tipo statico e tipo dinamico di un oggetto

Dopo l'assegnazione cb=ce, ogni uso di *preleva()* si riferisce alla sotto classe. Si dice che cb ha *tipo statico* (legato cioè alla dichiarazione) ContoBancario e *tipo dinamico* (guadagnato in seguito all'assegnazione) ContoConFido. Il tipo statico dice *cosa* si può fare su cb. Il tipo dinamico dice quale particolare metodo va in esecuzione, se uno della super classe o uno della sotto classe. Prima dell'assegnazione, cb.preleva(...) si riferisce al metodo della super classe. Dopo l'assegnazione, cb.preleva(...) invoca di fatto la versione di preleva di ContoConFido.

- su cb non posso chiamare i metodi aggiunti dalla sottoclasse, ma vanno comunque in funzione i metodi ridefiniti

Assegnazione dal generale al particolare ?



Il tipo statico mi consentirebbe di fare di più di ciò che è possibile sulla base del tipo dinamico

Non si può assegnare un oggetto dal generale al particolare, es. ce=cb. Tutto ciò si può subito comprendere riflettendo che cb non ha campi e valori corrispondenti ai campi particolari introdotti dalla classe conto con fido. Riferendoci nuovamente ad oggetti punto, non ha senso proiettare un punto dal piano cartesiano X-Y nello spazio, dal momento che non è definita la coordinata z.

Tuttavia, se cb ha tipo dinamico ContoConFido, si può di fatto cambiare *punto di vista* ("paio di occhiali") su cb in modo da *vederlo* come ContoConFido e quindi accedere a tutte le funzionalità di ContoConFido:

→ controlla il tipo dinamico

```
if( cb instanceof ContoConFido ){
   ce=(ContoConFido)cb; //casting
   ce.nuovoFido(5000);
   ...
}
```

cb continua ad essere di tipo ContoBancario e, tramite il casting è definito il nuovo oggetto ce

Su una variabile cb di classe (tipo statico) ContoBancario possono essere richieste sempre e solo le funzionalità della classe cui appartiene. Ma se cb ha tipo dinamico ContoConFido, invocando un metodo ridefinito in ContoConFido come *preleva/deposita*, di fatto si esegue la versione del metodo di ContoConFido. Se cb ha tipo dinamico ContoConFido (cosa controllabile con l'operatore **instanceof**) è possibile cambiare il

punto di vista su cb (*casting*) in modo da vederlo effettivamente come un oggetto di ContoConFido. Dopo il cambiamento di punto di vista si possono richiedere le funzionalità estese della sottoclasse.

Dunque:

ce=cb;

non ha senso e dà errore, ma

ce=(ContoEsteso)cb; Si può fare se il tipo dinamico di cb è ContoEsteso

è ok a meno che cb non disponga del tipo dinamico ContoConFido; tipicamente, per evitare l'eccezione ClassCastException, si fa precedere il test

```
if( cb instanceof ContoConFido ){
   su ((ContoConFido)cb) si possono
   richiedere operazioni secondo l'allargamento del
   punto di vista a ContoConFido
   eventualmente:
   ce=(ContoConFido)cb; etc.
}
```

Dynamic binding e polimorfismo

Il *dynamic binding* (collegamento dinamico) si riferisce alla proprietà che invocando un metodo su un oggetto come cb, dinamicamente possa essere eseguita la versione del metodo definita in ContoBancario o quella definita in ContoConFido, in dipendenza del tipo dinamico posseduto da cb.

Il termine *polimorfismo* significa "più forme" ed esprime la proprietà che un oggetto possa appartenere a più tipi. Assegnando cb=ce, l'oggetto cb acquisisce un altro tipo (diventa polimorfo). Il polimorfismo di cb si può verificare come segue

```
if( cb instanceof ContoBancario ) è TRUE
if( cb instanceof ContoEsteso ) è TRUE
```

A ben riflettere, dynamic binding e polimorfismo sono le due facce di una stessa medaglia. Proprio perché sussiste il polimorfismo, si ha l'effetto del dynamic binding

Ereditarietà e ridefinizione di metodi

Si è detto che il progetto di ContoConFido ridefinisce i metodi *deposita* e *preleva* già presenti nella super classe ContoBancario. Occorre prestare attenzione che per essere una vera ridefinizione, bisogna normalmente *rispettare* la sua intestazione (*signature*). Se cambia qualcosa nell'intestazione (nome del metodo, tipi dei parametri), allora si tratta di overloading anzichè di ridefinizione (overriding).

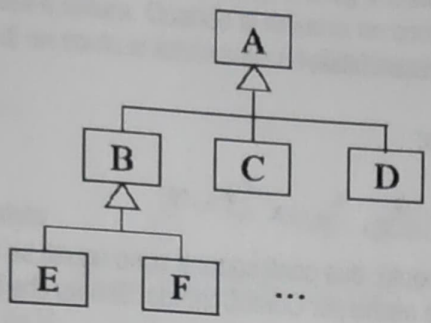
Perchè funzioni correttamente il dynamic binding/polimorfismo, è necessario osservare l'esatta intestazione dei metodi che, ad es., per *preleva* significa

```
@Override
public boolean preleva( double quanto ){...}
```

L'annotazione @Override disponibile dalla versione 5 di Java in poi, permette al compilatore di controllare ed eventualmente segnalare problemi, durante una ridefinizione. L'annotazione è facoltativa.

Ereditarietà singola

In Java ogni classe può essere erede di una sola classe (*ereditarietà singola*). Tutto ciò permette la costruzione di gerarchie di classi secondo una struttura ad albero, in cui ogni classe appartiene ad un solo percorso sino alla radice



L'esistenza di una gerarchia di classi, accresce le possibilità di polimorfismo. Ad es., oggetti di classe E hanno come tipi possibili: E, B ed A. Ciò è dovuto alla relazione di ereditarietà. Ad un oggetto di classe A è possibile assegnare un oggetto di una qualsiasi sottoclasse B, C, D, E, F ... dunque il tipo dinamico di un oggetto di classe A può essere una qualsiasi delle classi elencate.

Ereditarietà vs. composizione

Occorre sempre riflettere bene se una relazione di ereditarietà sia opportuna o se piuttosto non rappresenti una "forzatura", alla luce del principio di *sostituibilità dei tipi.*

Ad es. una Linea (segmento) è caratterizzata da due punti. Progettando la classe Linea come erede della classe Punto, contando sul fatto che un punto proviene dalla super classe, un altro lo si può aggiungere, si commette un errore grossolano. Infatti, una Linea *non* è un Punto piuttosto è *composta* (*has-a*) da due punti. Dunque anzichè ricorrere alla estensione, è opportuno programmare Linea come abbozzato di seguito:

```
class Linea{
   Punto p1, p2; //composizione mediante attributi
   ...
}
```

L'antenato "cosmico" Object

In Java, ogni classe eredita direttamente o indirettamente da **Object** (*radice* di tutte le gerarchie di classi). Quando una classe non specifica la clausola extends, in realtà ammette implicitamente la clausola

```
extends Object
```

La comune discendenza da Object si manifesta in diverse questioni, es. stile di programmazione, polimorfismo.

I seguenti metodi ammettono già un'implementazione in Object che necessariamente è generica. Essi vanno di norma ridefiniti per avere un significato "tagliato su misura" delle nuove classi:

- String toString() – ritorna lo stato di this sotto forma di stringa
- boolean equals(Object x) – ritorna true se this ed x sono uguali. Object definisce l'uguaglianza in modo "superficiale": due oggetti sono uguali se sono in aliasing, ossia condividono lo stesso riferimento
- int hashCode() – ritorna un hash code (numero intero unico) per this

Si mostra una ridefinizione del metodo equals() nella classe ContoBancario, che basa l'uguaglianza degli oggetti sul *contenuto* degli stessi (nozione "profonda" di uguaglianza):

```
@Override
public boolean equals( Object o ){
   if( !(o instanceof ContoBancario) ) return false;
   if( o==this ) return true;
   ContoBancario c=(ContoBancario)o;
   return numero.equals( c.numero );
}//equals
```

↳ equals della classe String

Il metodo si basa solo sul *numero* di conto: due conti correnti sono uguali se si riferiscono allo stesso numero di conto. Questa formulazione va bene anche per ContoConFido. Si nota che il test

```
if( !(o instanceof ContoBancario) ) return false;
```

include automaticamente anche il caso in cui **o** fosse null.

Strutture dati eterogenee

In virtù delle proprietà della relazione di ereditarietà, risulta ad es. che dichiarando

```
Object []v=new Object[10];
```

si possono memorizzare in v oggetti di qualunque classe concreta. L'array è generico ed eterogeneo. L'utente può comunque "scoprire" a runtime il tipo di un elemento con instanceof:

```
if( v[i] instanceof String ) ...
```

Riassunto modificatori

Gli attributi di una classe (campi o metodi) possono avere un modificatore tra

- **public** se sono esportati a tutti i possibili client
- **private** se rimangono ad uso esclusivo della classe
- **protected** se sono esportati solo alle classi eredi
- (*nulla*) se devono essere accessibili all'interno dello stesso package (familiarità o amicizia tra classi).

Attenzione: gli attributi protected sono accessibili *anche* nell'ambito del package di appartenenza.

Una classe può essere **public** se esportata per l'uso in altri file, non avere il modificatore **public** se l'uso della classe è ristretto al package (eventualmente anonimo) di appartenenza. Una classe può essere **final** se non può essere più estesa da classi eredi. Similmente, un metodo **final** non può essere più ridefinito nelle sottoclassi.

Gestione casi "anomali" (preliminare)

Si è visto che preleva(), deposita() ricevono una quantità che è attesa >0. Se così non è si potrebbe dare una segnalazione diagnostica e terminare il programma. In alternativa si può sollevare un'eccezione (il sistema delle eccezioni verrà approfondito più avanti nel corso) come segue:

```
public void deposita( double quanto ){
   if( quanto<=0 ) throw new IllegalArgumentException();
   ... //come prima
}//deposita
```