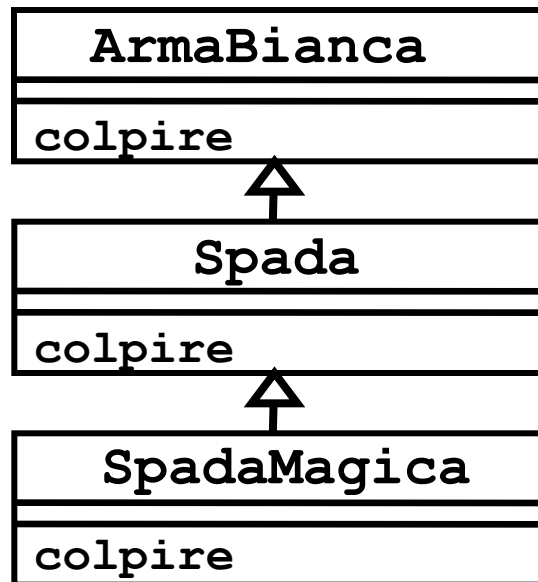


Polimorfismo

- Superclassi e sottoclassi sono costrutti distinti (le ultime poi ereditano metodi dalle prime).
- Però la potenza dell'ereditarietà sta nel consentire a oggetti delle sottoclassi di fungere da oggetti della superclasse: **polimorfismo**.
- Questo avviene grazie al fatto che in Java si accede agli oggetti tramite variabili riferimento.
- Un oggetto può essere riferito da una variabile della sottoclasse, o anche della superclasse.

Polimorfismo



- Un tipo numerico può sempre essere esteso a un altro tipo numerico se più generale.
 - ad esempio `int` → `double`.
- Analogamente un tipo riferimento può sempre essere convertito in un riferimento della superclasse.

```
SpadaMagica excalibur = new SpadaMagica();
Spada laSpadaNellaRoccia;
ArmaBianca lArmaDiReArtu;
laSpadaNellaRoccia = excalibur;
lArmaDiReArtu = excalibur;
```

assegnamenti
legittimi!

Polimorfismo

```
public void transfer(BankAccount b, double x)
{ this.withdraw(x); b.deposit(x); }
...
CheckingAccount contoDiLuca;
SavingsAccount fondoRisparmio;
contoDiLuca.transfer(fondoRisparmio, 1000);
```

- L'oggetto `contoDiLuca` eredita il metodo **`transfer()`** dalla superclasse **`BankAccount`**.
 - ▣ **`transfer()`** è invocato con un parametro più specializzato (**`SavingsAccount`** anziché **`BankAccount`**), ma il compilatore non si lamenta.

Polimorfismo

- In effetti, anche se il tipo del parametro non corrisponde, il compilatore copia il riferimento a **fondoRisparmio** (di tipo **SavingsAccount**, **sottoclasse**) in un riferimento a **b** (tipo **BankAccount**, **superclasse**).
- La conversione da sottoclasse a superclasse è lecita, non lo è tra classi non legate.

```
CheckingAccount x = fondoRisparmio; //NO
```

- Il compilatore sa che **CheckingAccount** non è una superclasse di **SavingsAccount**.

Polimorfismo

- E se, invece, riferiamo un oggetto della classe base con un riferimento di classe estesa?
 - ▣ Se l'oggetto è davvero della superclasse: ERRORE
- Ma supponiamo di avere scritto:
BankAccount x = fondoRisparmio;
- Più tardi, vogliamo sommare l'interesse al fondo **x**. Non possiamo trasformare il riferimento **x** di tipo **BankAccount** (superclasse) in uno di **SavingsAccount** (sottoclasse)? In fondo tutti questi riferimenti sono a oggetti di classe estesa

Polimorfismo

- Se siamo **assolutamente certi** che il riferimento **x** è per qualche motivo di tipo **BankAccount** ma l'oggetto che c'è dentro è di tipo **SavingsAccount**, possiamo fare un **cast**.
SavingsAccount y = (SavingsAccount)x;
- La notazione è la stessa per i cast numerici.
- Se ci sbagliamo, e l'oggetto **x** non è in realtà compatibile, il programma troverà un errore **a run time** e lancerà un'**eccezione**.

Polimorfismo

- A dire il vero c'è una differenza concettuale tra i cast numerici e i cast di tipi riferimento ereditati.
 - ▣ Per i tipi numerici, il cast è richiesto per dire al compilatore che siamo consapevoli che la conversione tra tipi numerici potenzialmente può farci perdere informazioni (es. **double** → **int**)
 - ▣ Per i tipi riferimento, stiamo dicendo al compilatore che siamo disposti a correre un rischio, e cioè che i tipi di cui effettuiamo il cast non siano in realtà legati da alcun rapporto di ereditarietà.

Operatore `instanceof`

- In generale, è meglio evitare i cast che non sono indice di buona programmazione. Alle volte però sono necessari. In tal caso potremmo volerci tutelare prima di farli.
- A questo scopo esiste l'operatore **`instanceof`** che controlla se un oggetto appartiene a una certa classe, ritornando un valore **`boolean`**:

 oggetto **`instanceof`** UnaClasse;

 ritorna **`true`** se oggetto può essere visto come istanza di **`UnaClasse`**, **`false`** altrimenti.

Operatore `instanceof`

- Quindi `x instanceof SavingsAccount`; vale `true` se `x` può essere visto come un oggetto `SavingsAccount`, `false` altrimenti.
- Un cast sicuro è ottenuto come segue:

```
SavingsAccount y;  
if (x instanceof SavingsAccount)  
    y = (SavingsAccount)x;  
else  
    y = null;
```

Polimorfismo

- Torniamo a questo esempio.

```
SpadaMagica excalibur = new SpadaMagica();  
Spada laSpadaNellaRoccia;  
ArmaBianca lArmaDiReArtu;
```

```
laSpadaNellaRoccia = excalibur;  
lArmaDiReArtu = excalibur;
```

assegnamenti
legittimi!

```
excalibur.colpire();  
laSpadaNellaRoccia.colpire();  
lArmaDiReArtu.colpire();
```

quale metodo
viene eseguito?

- L'oggetto **excalibur** può forse **colpire()** in più di un modo?

Polimorfismo

- In realtà durante l'**esecuzione** saranno verificate delle promozioni implicite.
 - ▣ La JVM guarda di che tipo è davvero l'oggetto (non il riferimento) e usa il metodo più appropriato.
- Per l'istruzione **unArma.colpire()** ;
 - ▣ se la JVM vede che **unArma** è un oggetto **SpadaMagica**, invoca il metodo corrispondente
 - ▣ altrimenti, se è una **Spada**, usa il metodo di **Spada**
 - ▣ altrimenti userà il metodo di **ArmaBianca**
- Si sale nella gerarchia fino eventualmente a **Object**

Tipi statici e dinamici

- Quindi le **variabili** riferimento e gli **oggetti** che esse riferiscono hanno rispettivamente:
 - ▣ un **tipo statico**, determinato dal compilatore
 - ▣ un **tipo dinamico**, valutato a run-time dalla JVM
- In compilazione viene valutato il tipo statico
 - ▣ ad es.: gli assegnamenti sono compatibili? cast?
- In fase di esecuzione il tipo di un oggetto può differire da quello atteso sulla base della sua variabile riferimento.
 - ▣ Nota: per i tipi numerici statico = dinamico sempre!

Tipi statici e dinamici

- Il tipo statico è assegnato quando la variabile viene **dichiarata**. Il tipo dinamico è invece assegnato quando viene usato il **costruttore**.

Pertanto, è perfettamente lecito scrivere:

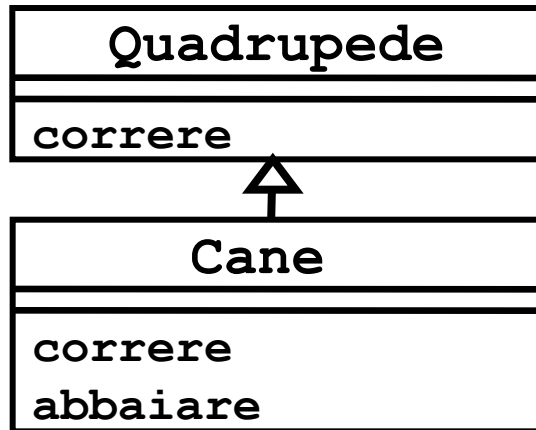
```
ClasseBase var = new ClasseEstesa();
```

variabile riferimento della
superclasse **ClasseBase**

oggetto costruito dal costruttore
della sottoclasse **ClasseEstesa**

solo, tipo **statico** e **dinamico** saranno diversi.

Tipi statici e dinamici



```
Cane fido = new Cane();
Quadrupede myPet = fido;
fido.abbaia();
myPet.abbaia(); //NO!
```

- Otteniamo un errore **in compilazione**.
- Le variabili `fido` e `myPet` puntano lo stesso oggetto di tipo `Cane`, ma `fido` è pure **dichiarata** come `Cane`, mentre è **dichiarata** come `Quadrupede` (suo tipo statico) e quindi non ha un metodo `abbaiare()`.

Late binding

- Si parla di ***early binding*** e ***late binding*** (letteralm.: primo legame e legame ritardato)
- L'early binding è il legame in compilazione tra una variabile e il suo tipo statico
 - ▣ A causa dell'early binding, non si può invocare un metodo non previsto per oggetti di quel tipo statico
- Il late binding invece si basa sul tipo dinamico e decide qual è il metodo giusto da usare
 - ▣ Determina l'**effettivo comportamento** dell'oggetto

Shadowing

- Se si prova a fare “overriding” di una variabile, si ottiene un altro effetto. La variabile della superclasse viene “messa in ombra” da quella della sottoclasse.
- Mettere in ombra una variabile è un errore comune. L'effetto è simile a quello già visto per le definizioni di variabili locali in un metodo con lo stesso nome di una variabile d'istanza o di una variabile di classe.

Variabili in ombra

- Se vengono messe in ombra variabili d'istanza, si creano comportamenti ambigui.
- Ad esempio abbiamo visto che non si poteva accedere **balance** dentro **CheckingAccount**.

```
class CheckingAccount extends BankAccount
{ //campi
  ...
  //metodi
  public void deposit(double x)
  { ... balance += x; // ERRORE!
  }
}
```

Variabili in ombra

- Un modo (sbagliato) di “risolvere” il problema è quello di definire una **nuova** variabile `balance`. In realtà così facendo si mette in ombra quella della superclasse.

```
class CheckingAccount extends BankAccount
{ //campi
  double balance;
  //metodi
  public void deposit(double x)
  { ... balance += x; // NON SERVE! }
}
```

non viene toccato
(è messo in ombra)

balance (CA)

balance (BA)

valore
modificato

Variabili in ombra

- La cosa va ancora peggio se si usano tutte variabili pubbliche: si hanno ambiguità.

```
public class classeBase
{ public int var;
  public classeBase() { var = 0; }
  public getVar() { return var; }
}

public class classeEstesa extends ClasseBase
{ public int var;
  public classeEstesa() { var = 8; }
  public getVar() { return var; }
}
```

Variabili in ombra

```
classeBase a = new classeEstesa();
```

- Si usa il costruttore di **classeEstesa** per creare un oggetto, ma il riferimento è a **classeBase**. Il metodo **getVar** è dominato (sotto overriding), il campo **var** è in ombra.

a: classeEstesa

public int var	8
public int (hidden) var	0

int getVar()

due campi, di cui uno in ombra

un solo metodo

Variabili in ombra

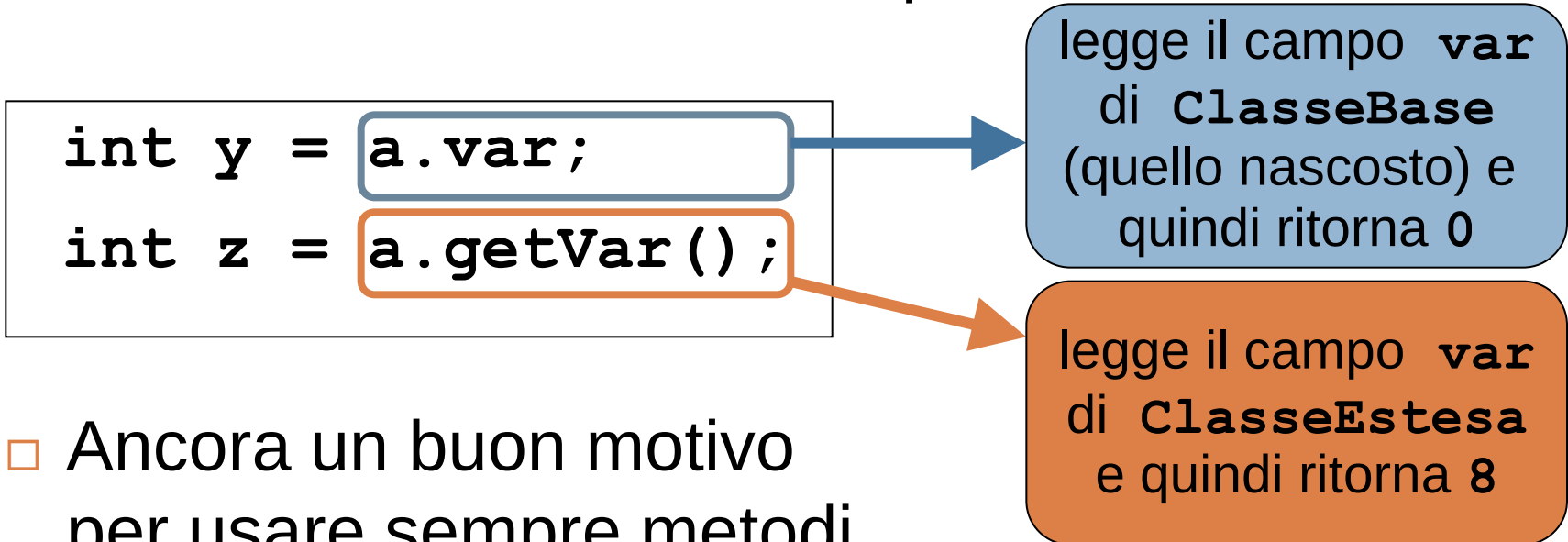
```
classeBase a = new classeEstesa();
```

- Quindi l'oggetto **a** contiene due campi entrambi chiamati **var**. Attenzione però, perché sebbene l'oggetto **a** sia di tipo dinamico **classeEstesa**, il suo riferimento è memorizzato col tipo statico **classeBase**.
 - ▣ I campi dell'oggetto vengono acceduti tramite il suo riferimento (e quindi il tipo statico).
 - ▣ Questo dice come la cella di memoria va "letta".

Variabili in ombra

- Perciò, se si prova a leggere **var** dando come riferimento **a** si ha un valore diverso a seconda che si usi un campo o un metodo.

```
int y = a.var;  
int z = a.getVar();
```



legge il campo **var**
di **ClasseBase**
(quello nascosto) e
quindi ritorna 0

legge il campo **var**
di **ClasseEstesa**
e quindi ritorna 8

- Ancora un buon motivo per usare sempre metodi di accesso e campi privati.

Polimorfismo e overloading

- ❑ Con l'overloading è possibile definire metodi con lo stesso **nome** ma con **firma** differente.
- ❑ Ovvero, hanno parametri espliciti di tipi diversi.
- ❑ Ma è davvero così? Il compilatore può garantire che il **tipo statico** dei parametri sia lo stesso, ma non il loro **tipo dinamico**.
- ❑ Potremmo invocare anche un metodo pur in assenza di una perfetta corrispondenza di tipi dei parametri, contando su un cast a run-time

Polimorfismo e overloading

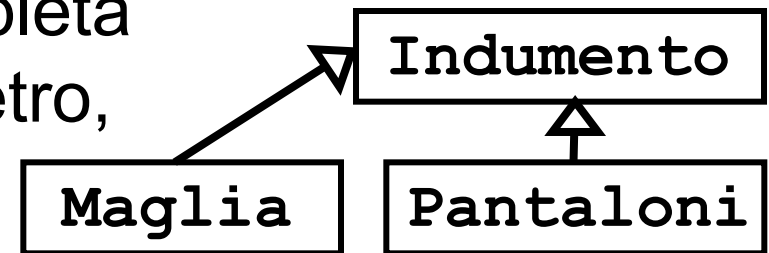
- Il compilatore si preoccupa di verificare che i parametri attuali corrispondano, e risolve l'overloading decidendo qual è il metodo giusto da usare tra più metodi omonimi.
 - ▣ Questo è meno semplice del previsto se i metodi coi parametri che corrispondono sono più d'uno (ad esempio, perché ci sono metodi con parametri meno specifici, ma pur sempre compatibili)
- Possono insorgere delle **ambiguità**.

Polimorfismo e overloading

- Il compilatore deve identificare un metodo con uno specifico bytecode, perciò risolve le potenziali ambiguità come segue:
 - ▣ se tra i metodi possibili solo uno ha i riferimenti dei parametri attuali che possono corrispondere ai parametri formali, sceglie quello
 - ▣ altrimenti, prende **il più specifico** di quelli disponibili (in particolare: se c'è uno che corrisponde perfettamente, sceglie quello)
 - ▣ se anche così l'ambiguità persiste, il compilatore **ritorna un errore**

Polimorfismo e overloading

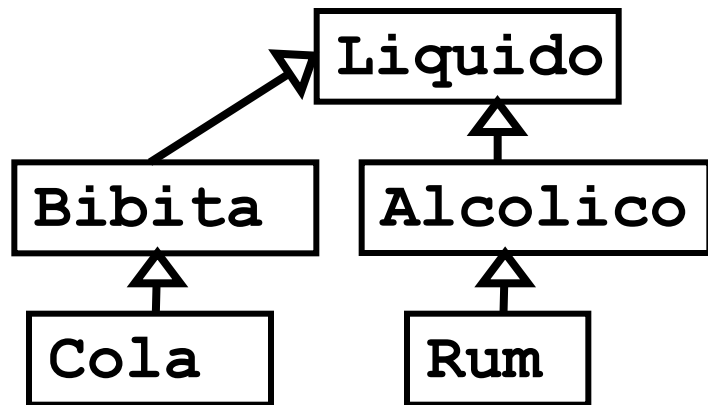
- La relazione “è più specifico di” è una relazione d'ordine completa solo se c'è un unico parametro, altrimenti è incompleta.



Metodo 1: `indossa(Maglia a, Pantaloni b)`
Metodo 2: `indossa(Maglia a, Indumento b)`
Metodo 3: `indossa(Indumento a, Pantaloni b)`
Metodo 4: `indossa(Indumento a, Indumento b)`

- Il metodo 1 è più specifico di tutti. Ma tra i metodi 2 e 3 nessuno è più specifico dell'altro

Esempio



Un oggetto ha i metodi:

mischia(Liquido l, Alcolico a)

mischia(Bibita b, Liquido l)

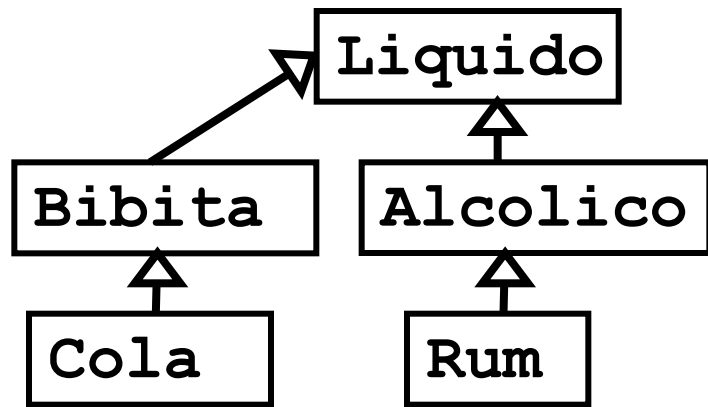
mischia(Cola c, Alcolico a)

ed esistono le variabili riferimento:

rL, rB, rA, rC, rR

- Se invochiamo: **mischia(rL, rA)** ;
viene utilizzato il **primo metodo** perché
parametri formali e attuali **corrispondono
perfettamente**.

Esempio



Un oggetto ha i metodi:

mischia(Liquido l, Alcolico a)

mischia(Bibita b, Liquido l)

mischia(Cola c, Alcolico a)

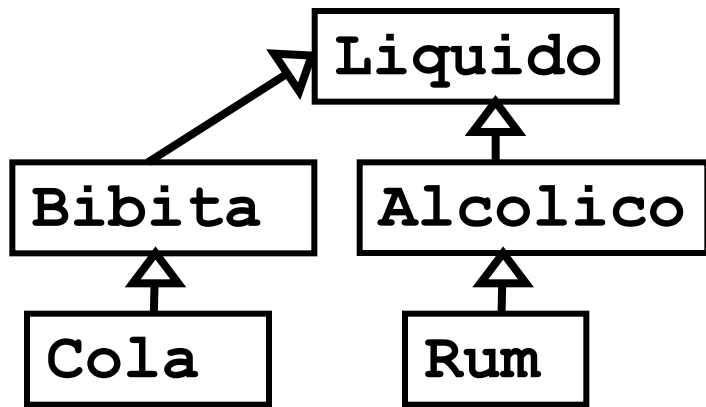
ed esistono le variabili riferimento:

rL, rB, rA, rC, rR

□ Se invochiamo: **mischia(rC, rL)** ;

viene utilizzato il **secondo metodo** perché è **il solo** che può corrispondere (facendo un cast di **rC** a **Bibita**, che è ammesso)

Esempio



Un oggetto ha i metodi:

mischia(Liquido l, Alcolico a)

mischia(Bibita b, Liquido l)

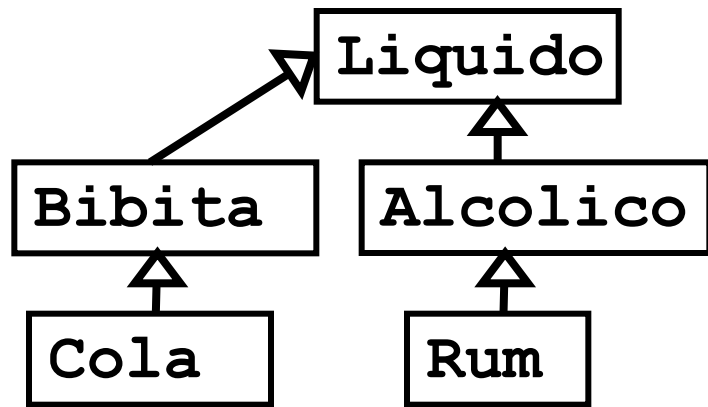
mischia(Cola c, Alcolico a)

ed esistono le variabili riferimento:

rL, rB, rA, rC, rR

- Se invochiamo: **mischia(rC, rR)** ;
 - ▣ nessun metodo corrisponde perfettamente
 - ▣ tutti e tre i metodi corrispondono dopo un cast
 - ▣ tuttavia il **terzo metodo** è quello **più specifico**

Esempio



Un oggetto ha i metodi:

mischia(Liquido l, Alcolico a)

mischia(Bibita b, Liquido l)

mischia(Cola c, Alcolico a)

ed esistono le variabili riferimento:

rL, rB, rA, rC, rR

- Se invochiamo: **mischia(rB, rA)** ;
 - ▣ il terzo metodo non corrisponde
 - ▣ il primo e il secondo corrispondono ma non perfettamente e nessuno **più specifico** dell'altro
 - ▣ quindi il compilatore si pianta e segnala un **errore**

Precisazioni

- Le regole si applicano allo stesso modo anche se alcuni dei parametri sono tipi primitivi.
- La risoluzione dell'overloading **NON** usa:
 - ▣ il tipo di ritorno del metodo, perché il compilatore si basa solo sulla firma, non sul prototipo.
 - ▣ le eccezioni che il metodo prova a lanciare, perché queste hanno senso solo a run-time, ma l'overloading è risolto in compilazione

Late binding

- Se c'è sia overloading che overriding:
 - ▣ il compilatore esegue un early binding per capire qual è il metodo da usare tra quelli che hanno la stessa firma
 - ▣ la decisione su qual è il metodo effettivo da usare, tra quelli con la stessa firma, viene effettuata a run-time basandosi sul tipo dinamico degli oggetti (late binding)

Modificatori di visibilità

- Attenzione nell'overriding dei metodi: la restrizione di visibilità può essere **allentata**, ma mai resa più restrittiva.
- In particolare non si può ridefinire come privato un metodo che era inizialmente pubblico.

```
public class classeBase
{ public operazione() { ... }
}

public class classeEstesa extends ClasseBase
{ private operazione() { ... } //ERRORE!
}
```



Modificatori di visibilità

- Il compilatore segnala errore perché dare visibilità ridotta al metodo `operazione()` di `ClasseEstesa` è illusorio.

```
this.operazione();    //PRIVATO, ma...  
super.operazione(); //PUBBLICO
```



- Il metodo ridefinito non può essere invocato da altre classi, quello di partenza resta pubblico.
 - Si viola il principio per cui un'istanza della sottoclasse è anche istanza della superclasse (e quindi può fare più cose, non meno).

Esempio: BackupCell

- La classe **Cell** rappresenta una cella di memoria di un intero, con opportuni metodi

```
public class Cell
{ //campi
  private int val;
  //costruttori
  public Cell(int v) { val = v; }
  public Cell() { this(0.0); }
  //metodi
  public int getVal() { return val; }
  public void setVal(int v) { val=v; }
  public void clear() { val=0; }
}
```

Esempio: BackupCell

- Esercizio: estendere la classe a una classe **BackupCell**, che fa le stesse cose ma in più fornisce una funzionalità di backup. Ovvero:
 - esiste un **campo** **oldVal** in più per salvare il vecchio valore tutte le volte che c'è una modifica
 - i metodi esistenti vanno sovrascritti in qualche modo per tenere conto di questa funzionalità
 - esisterà un metodo aggiuntivo **restore()** per ripristinare il valore di backup

Esempio: AccountTelefono

- Sia definita una classe **AccountTelefonico**, che riassume le caratteristiche di più tipologie di contratti per linea telefonica. La classe ha:
 - ▣ campi: numero di telefono, nome dell'abbonato, numero di telefonate effettuate, durata in secondi delle telefonate effettuate, e (valore costante e valido per tutta la classe) un costo al secondo; usare campi final dove necessario
 - ▣ metodi: per accedere e settare i valori, nei modi che si ritiene necessario

Esempio: AccountTelefono

- Vogliamo estendere la classe alle sottoclassi:
 - ▣ **AccountFisso**, che modella un account di linea fissa: contiene in più un campo per l'indirizzo
 - ▣ **AccountMobile**, che modella un account cellulare: ad ogni chiamata aggiunge un costo pari alla costante **SCATTO** (scatto alla risposta)
 - ▣ **AccountYouAndMe**, che è un caso particolare di account cellulare dove in aggiunta c'è un campo di numero you-and-me. Le chiamate verso questo numero sono gratis.

Esempio: AccountTelefono

- Prima, rappresentare le classi in diagramma
- Poi, vanno implementati questi metodi:
 - ▣ accesso: per avere i dati dell'utente
 - ▣ aggiornare la bolletta se è stata effettuata una telefonata di N secondi
 - ▣ aggiornare la bolletta se è stata effettuata una telefonata di N secondi a un certo numero (metodo sovraccarico)
 - ▣ ottenere l'ammontare speso finora dall'utente
 - ▣ ottenere il numero delle telefonate fatte