Ereditarietà e Polimorfismo

Riusare il software

- A volte si incontrano classi con funzionalità simili

  In quanto sottendono concetti semanticamente “vicini”

- È possibile creare classi disgiunte replicando le porzione di stato/comportamento condivise

  L’approccio “Taglia&Incolla”, però, non è una strategia vincente

  Difficoltà di manutenzione correttiva e perfettiva

- Meglio “specializzare” codice funzionante

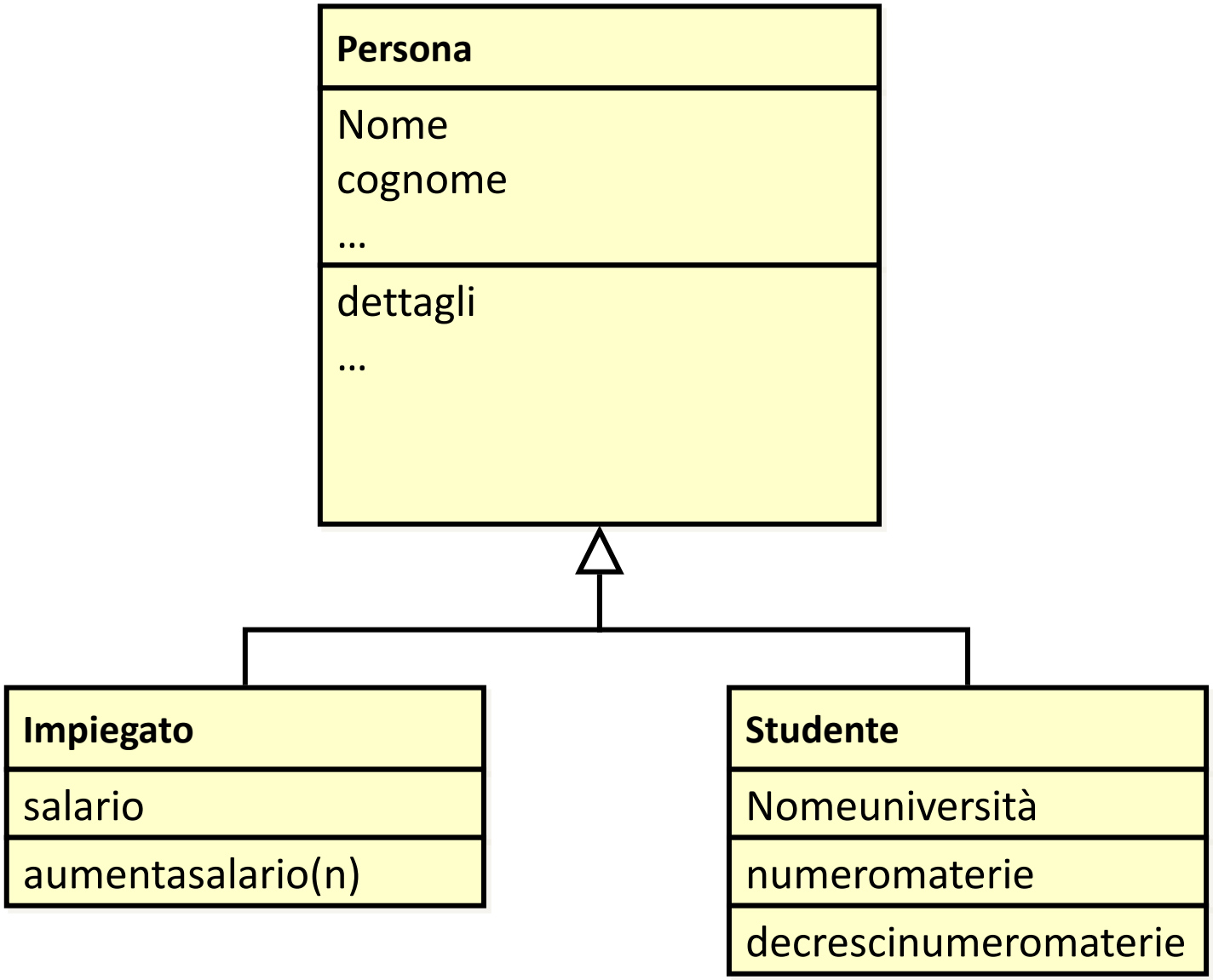
  Sostituendo il minimo necessario

Ereditarietà

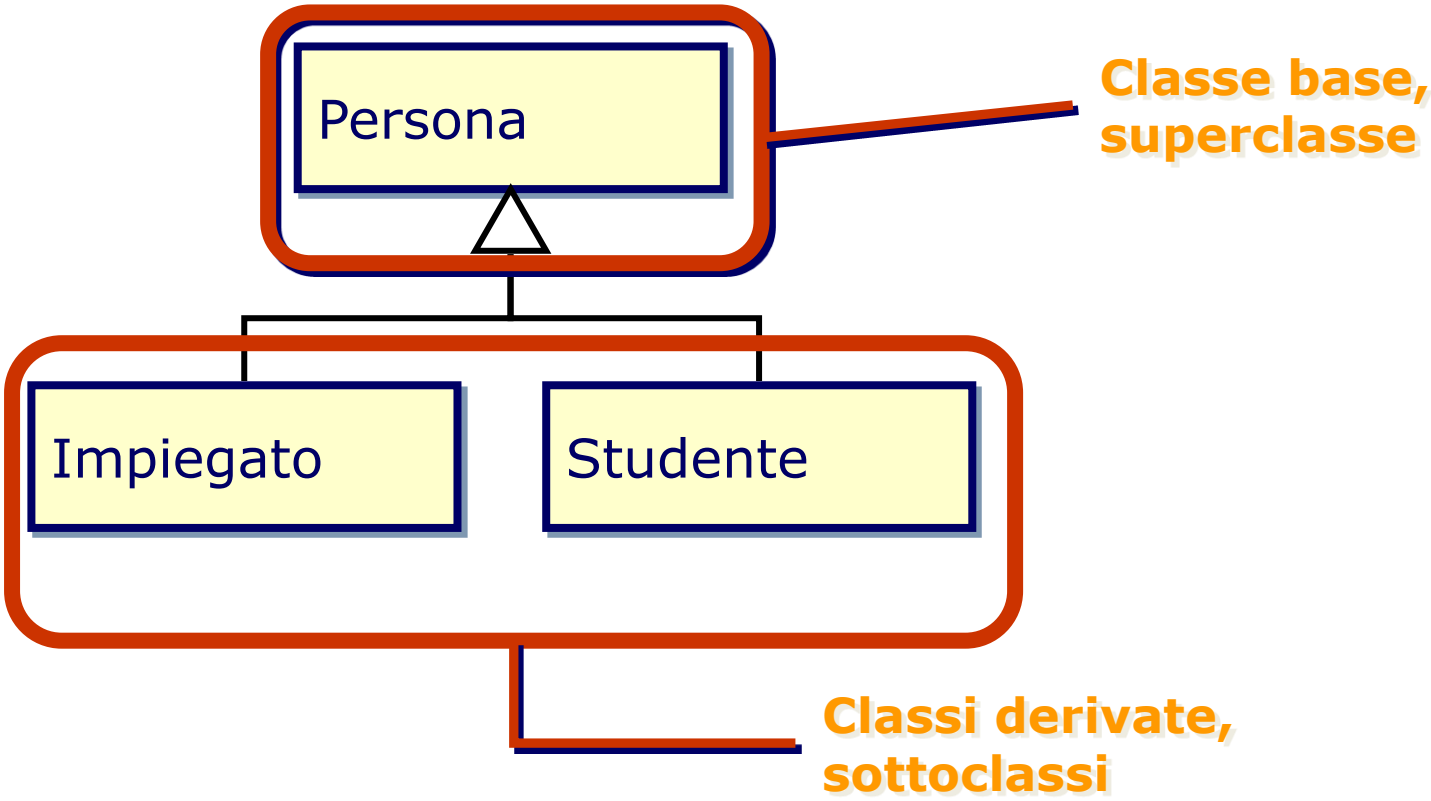
- Meccanismo per definire una nuova classe (classe derivata) come specializzazione di un'altra (classe base)
  - La classe **base** modella un concetto **generico**
  - La classe **derivata** modella un concetto **più specifico**

- La classe derivata:
  - Dispone di tutte le funzionalità (attributi e metodi) di quella base
  - Può aggiungere funzionalità proprie
  - Può ridefinirne il funzionamento di metodi esistenti (polimorfismo)

Esempio


Persona
Nome
cognome
…
dettagli
…
Impiegato
salario
aumentasalario(n)
Studente
Nomeuniversità
numeromaterie
decrescinumeromaterie

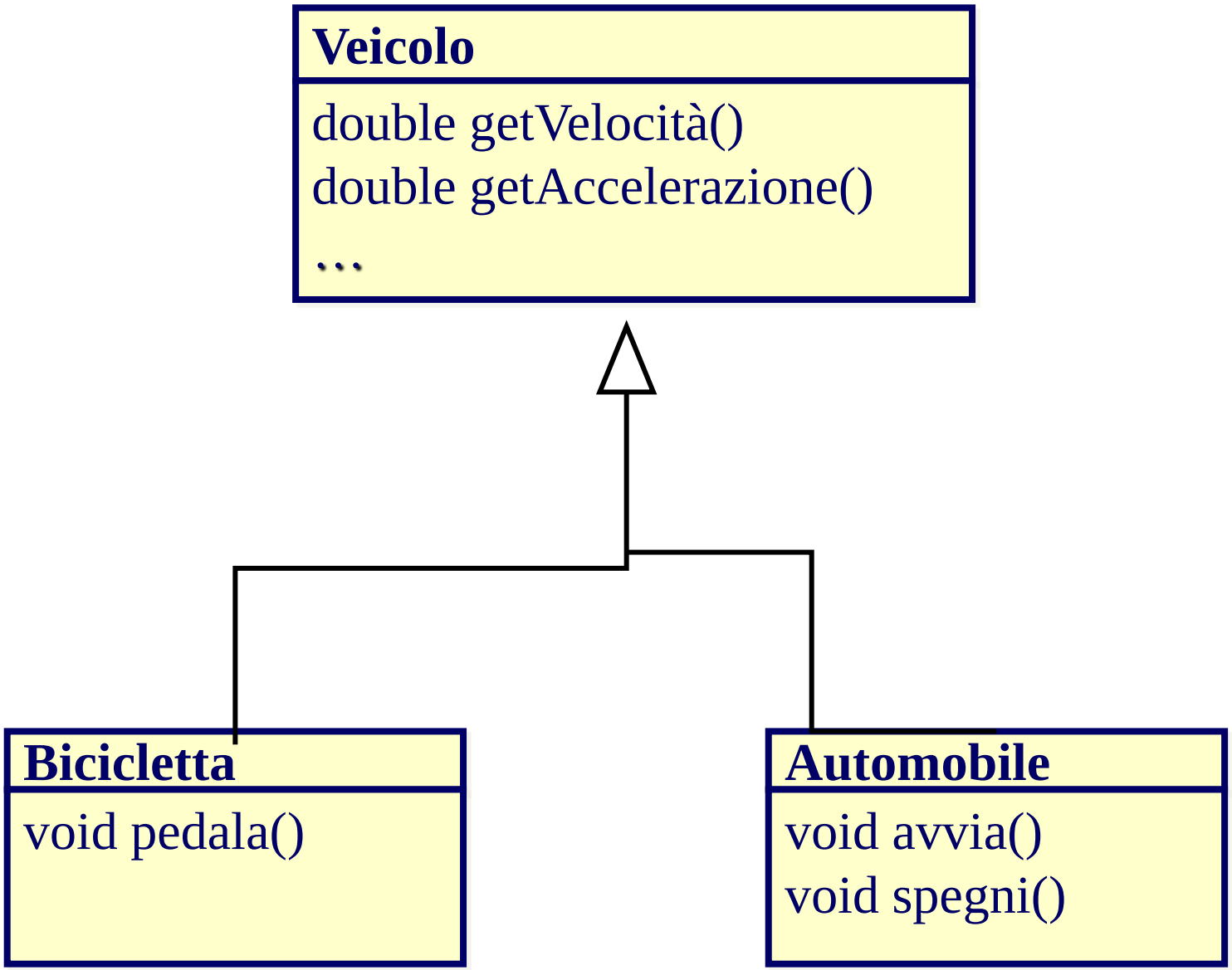
Terminologia



Astrazione

- Il processo di analisi e progettazione del software di solito procede per raffinamenti successivi
  - Spesso capita che le similitudini tra classi non siano colte inizialmente
  - In una fase successiva, si coglie l'esigenza/opportunità di introdurre un concetto più generico da cui derivare classi specifiche

- Processo di astrazione
  - Si introduce la superclasse che "astrae" il concetto comune condiviso dalle diverse sottoclassi
  - Le sottoclassi vengono "spogliate" delle funzionalità comuni che migrano nella superclasse


Veicolo
double getVelocità()
double getAccelerazione()
…
Bicicletta
void pedala()
Automobile
void avvia()
void spegni()

Tipi ed ereditarietà

- Ogni classe definisce un tipo:
  - Un oggetto, istanza di una sotto-classe, è **formalmente** compatibile con il tipo della classe base
  - Il contrario non è vero!
- Esempio
  - Un'automobile è un veicolo
  - Un veicolo non è (necessariamente) un'automobile
- La compatibilità diviene effettiva se
  - I metodi ridefiniti nella sotto-classe rispettano la semantica della superclasse
- L'ereditarietà gode delle proprietà transitiva
  - Un tandem è un veicolo (poiché è una bicicletta, che a sua volta è un veicolo)

Vantaggi dell'ereditarietà

- Evitare la duplicazione di codice
- Permettere il riuso di funzionalità
- Semplificare la costruzione di nuove classi
- Facilitare la manutenzione
- Garantire la consistenza delle interfacce

Ereditarietà in Java

- Si definisce una classe derivata attraverso la parola chiave "**extends**"
  - Seguita dal nome della classe base

- Gli oggetti della classe derivata sono, a tutti gli effetti, estensioni della classe base
  - Anche nella loro rappresentazione in memoria

Ereditarietà in Java

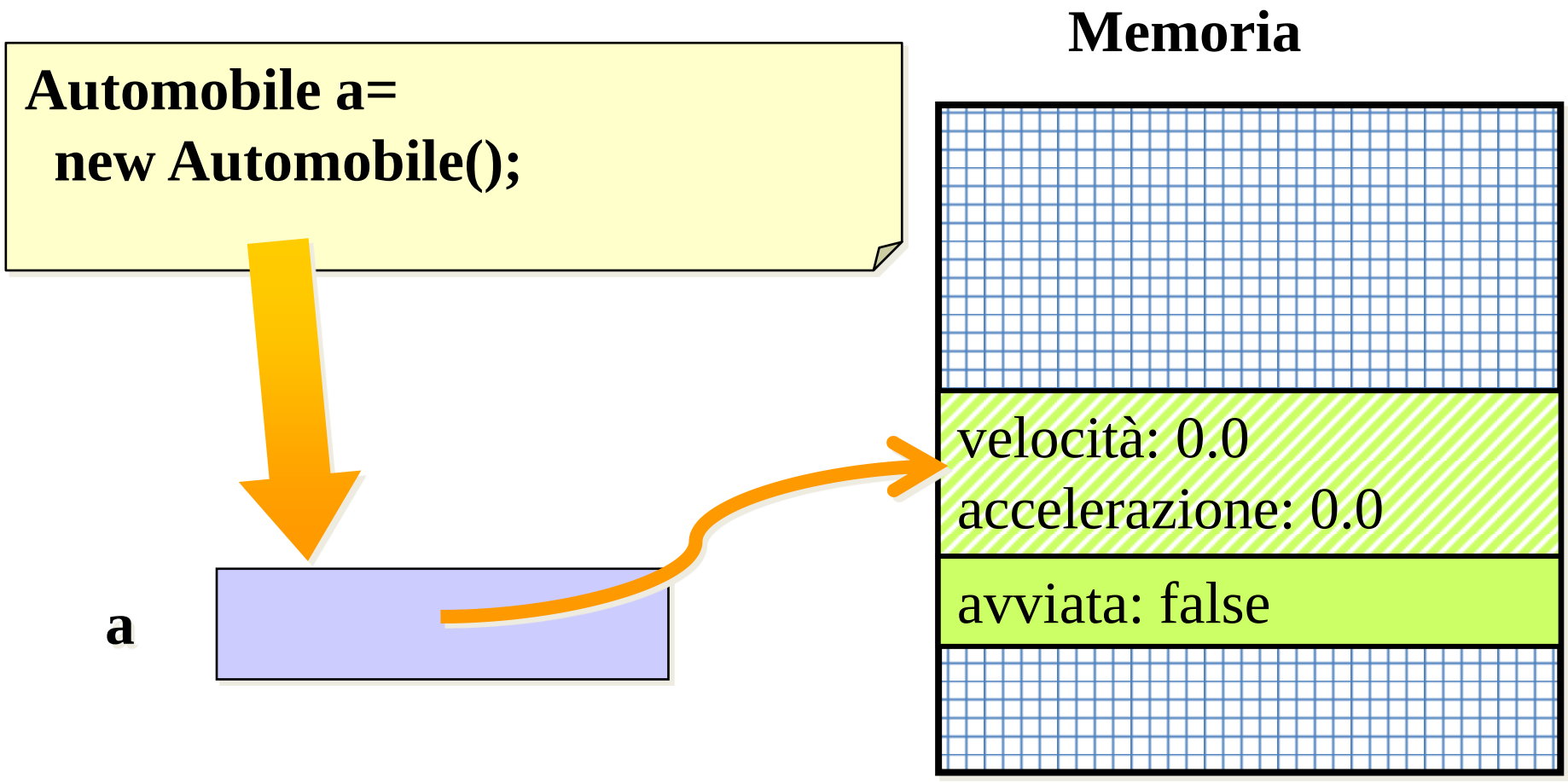
Veicolo.java

```
public class Veicolo {
        private double velocità;
        private double accelerazione;
        public double getVelocità() {…}
        public double getAccelerazione() {…}
}
```

Automobile.java

```
public class Automobile
    extends Veicolo {
          private boolean avviata;
          public void avvia() {…}
}
```

Ereditarietà in Java



Meccanismi

- Costruzione di oggetti di classi derivate
- Accesso alle funzionalità della superclasse
- Ri-definizione di metodi

Costruttori

- Per realizzare un'istanza di una classe derivata, occorre – innanzi tutto – costruire l'oggetto base
  - Di solito, provvede automaticamente il compilatore, invocando – come prima operazione di ogni costruttore della classe derivata – il costruttore anonimo della superclasse
  - Si può effettuare in modo esplicito, attraverso il costrutto ***super(…)***
  - Eventuali ulteriori inizializzazioni possono essere effettuate solo successivamente

Esempio

```
class Impiegato {
  String nome;
  double stipendio;

  Impiegato(String n) {
        nome = n;
        stipendio= 1500;
  }
}
```

```
class Funzionario
  extends Impiegato {

    Funzionario(String n) {
        super(n);
        stipendio = 2000;
  }
}
```

Accedere alla superclasse

- L'oggetto derivato contiene **tutti i componenti** (attributi e metodi) dell'oggetto da cui deriva
  - Ma i suoi metodi **non possono** operare direttamente su quelli definiti **privati**

- La restrizione può essere allentata:
  - La super-classe può definire attributi e metodi con visibilità "**protected**"
  - Questi sono visibili alle sottoclassi

Ridefinire i metodi

- Una sottoclasse può ridefinire metodi presenti nella superclasse

- A condizione che abbiano
  - Lo stesso **nome**
  - Gli stessi **parametri** (tipo, numero, ordine)
  - Lo stesso **tipo di ritorno**
  - (La stessa **semantica**!)

- Per le istanze della sottoclasse, il nuovo metodo **nasconde** l'originale

Ridefinire i metodi

```
class Base {
        int m() {
         return 0;
     }
}
```

```
class Derivata
  extends Base {
        int m() {
         return 1;
     }
}
```

```
Base b= new Base();
System.out.println(b.m());
Derivata d= new Derivata();
System.out.println(d.m());
```

Ridefinire i metodi

- A volte, una sottoclasse vuole "perfezionare" un metodo ereditato, non sostituirlo *in toto*
  - Per invocare l'implementazione presente nella super-classe, si usa il costrutto
    super.<*nomeMetodo*> (…)

```
class Base {
        int m() {
         return 0;
     }
}
```

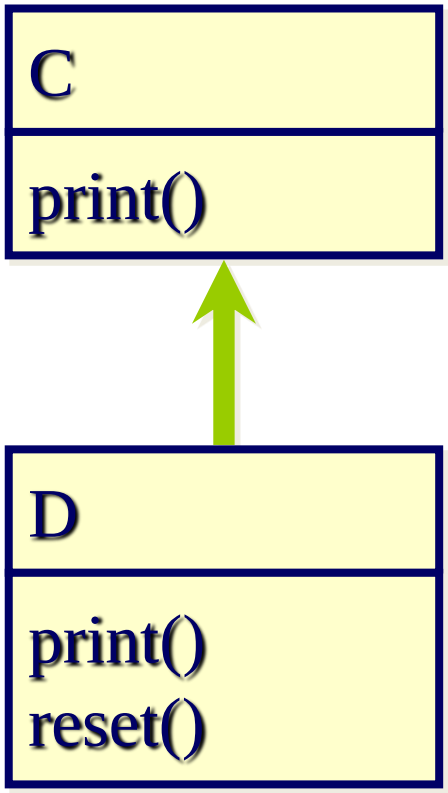
```
class Derivata
        extends Base {
        int m() {
            return super.m()+ 1;
     }
}
```

Compatibilità formale

- Un'istanza di una classe derivata è formalmente compatibile con il tipo della super-classe
  - Base b = new Derivata( );

- Il tipo della variabile "b" (Base) limita le operazioni che possono essere eseguite sull'oggetto contenuto
  - Anche se questo ha una classe più specifica (Derivata), in grado di offrire un maggior numero di operazioni
  - Altrimenti viene generato un errore di compilazione

Compatibilità formale




```
C v1= new C();
C v2= new D();
D v3= new D();
```

v1.print() ☑
v2.print() ☑
v2.reset() ☒
v3.reset() ☑

Polimorfismo

```
class Base {
        int m() {
         return 0;
     }
}
```

```
class Derivata
  extends Base {
        int m() {
         return 1;
     }
}
```

```
Base b= new Derivata();
System.out.println(b.m());
```

Polimorfismo

- Java mantiene traccia della classe effettiva di un dato oggetto
  - Seleziona sempre il metodo più specifico…
  - …anche se la variabile che lo contiene appartiene ad una classe più generica!

- Una variabile generica può avere "molte forme"
  - Contenere oggetti di **sottoclassi differenti**
  - In caso di ridefinizione, il metodo chiamato dipende dal **tipo effettivo** dell'oggetto

Polimorfismo

- Per sfruttare questa tecnica:
  - Si definiscono, nella super-classe, metodi con implementazione generica…
  - …sostituiti, nelle sottoclassi, da implementazioni specifiche
  - Si utilizzano variabili aventi come tipo quello della super-classe

- Meccanismo estremamente potente e versatile, alla base di molti "pattern" di programmazione

Esercizio 5

Create il **tipo di dato Counter** dell'Esercizio 1 come **sottoclasse del tipo di dato SimpleCounter.**

Esercizio 1

Create un **tipo di dato Counter** che abbia:

un valore attuale
un valore massimo di conteggio
uno stato interno che indica se si è verificato un errore. Il valore massimo è selezionabile dall'utente alla costruzione. Se si tenta di superarlo, viene modificato lo stato del contatore per memorizzare l'avvenuta condizione d'errore e le successive operazioni di modifica non devono avere effetto

Il contatore deve offrire **i seguenti metodi:** void inc(),void reset(),int getValue(),boolean isError()

Infine, scrivete un'applicazione Java che:

Crea e inizializza un nuovo Counter con valore massimo n
Incrementa il contatore per n+1 volte e visualizza, ogni volta, il valore attuale e lo stato interno (errato/corretto)
Terminare

Esercizio 6

Create il **tipo di dato Impiegato** come estensione del **tipo di dato Persona.**

Dove una **classe Persona** ha le variabili nome, cognome, età. Ha i metodi GetNome, GetCognome, GetEtà e un metodo dettagli() che restituisce in una stringa le informazioni sulla persona in questione.

Dove una **classe Impiegato** ha le variabili nome, cognome, età, salario. Ha un metodo dettagli() che restituisce in una stringa le informazioni sulla persona in questione.
Ha un metodo aumentasalario() che aumenti lo stipendio secondo una certa percentuale.

Inoltre, per entrambi gli esercizi scrivere un main con le istruzioni presenti rispettivamente nell'esercizio 1 e nell'esercizio 4.