UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Università di degli Studi di Salerno
Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2023-2024

Ereditarietà

Docente: Prof. Massimo Ficco

E-mail: *mficco@unisa.it*

Riuso del codice

Composizione

- Qualcosa di già visto
- Gli attributi della nostra classe sono oggetti di classi già esistenti (della VM o create da noi)

Ereditarietà

È una dei meccanismi fondamentali della programmazione ad oggetti

Composizione

Classi componenti

```
class Engine {
   public void start() {}
   public void rev() {}
   public void stop() {}
}
class Wheel {
    public void inflate(int psi) {}
}
class Window {
   public void rollup() {}
   public void rolldown() {}
}
```

```
class Door {
  public Window window =
          new Window();
  public void open() {}
  public void close() {}
}
```

Classe principale

```
public class Car {
        public Engine engine = new Engine();
        public Wheel[] wheel = new Wheel[4];
        public Door left , right;    // 2-door
        public Car() {
                left = new Door(); right = new Door();
                for(int i = 0; i < 4; i++)
                        wheel[i] = new Wheel();
        }

public static void main(String[] args) {
     Car car = new Car();
     car.left.window.rollup();
     car.wheel[0].inflate(72);
  }
}
```

Inizializzazione dei componenti

Occorre fare attenzione ad inizializzare gli oggetti componenti di una classe:

- Nella dichiarazione
- Nel costruttore della classe
- Appena prima di essere usati

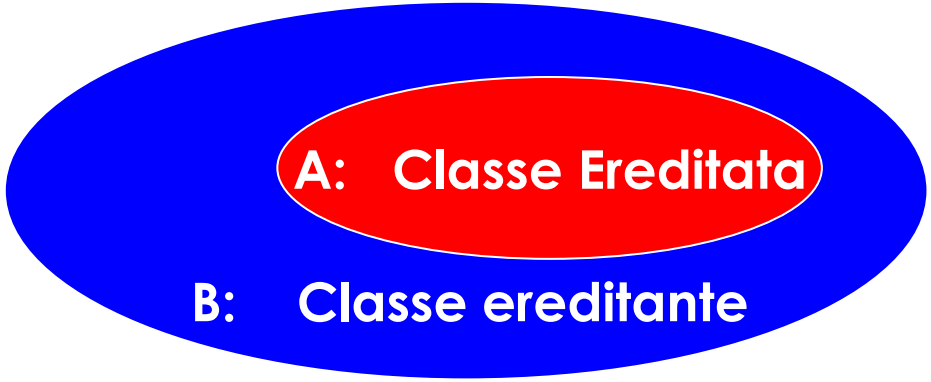
Ereditarietà

Ereditarietà

La relazione di ereditarietà equivale alla relazione di inclusione tra gli insiemi.

Dire che una classe **B** eredita un'altra **A** equivale a dire che **B** ha sicuramente tutti gli attributi ed i metodi di **A.**

Ereditando **A** possiamo estendere la sua definizione completando **B**



Indicazioni d'uso !

Si **usa la composizione** quando una classe deve fornire le funzionalità realizzate in altre classi già esistenti

Si **usa l'ereditarietà** quando la nuova classe deve presentare un estensione dell'interfaccia della vecchia classe

Esempio

Classe veicolo:

- Pilota
- Ruote

- **Classe Auto**
  - Eredita veicolo
  - Volante
  - Motore
  - Targa

- **Classe Bici**
  - Eredita veicolo
  - Manubrio
  - Pedali

- **Classe Autotreno**
  - Eredita auto
  - Rimorchio

Ereditarietà

Esiste però anche un altro motivo, di ordine pratico, per cui conviene usare l'ereditarietà, oltre quello di descrivere un sistema secondo un modello gerarchico; questo secondo motivo è legato esclusivamente al concetto di **riuso del software**

In alcuni casi si ha a disposizione una classe che non corrisponde esattamente alle proprie esigenze. Anziché scartare del tutto il codice esistente e riscriverlo, si può seguire con l'ereditarietà un approccio diverso, costruendo una nuova classe che eredita il comportamento di quella esistente, salvo che per i cambiamenti che si ritiene necessario apportare

Tali cambiamenti possono riguardare sia l'aggiunta di nuove funzionalità che la modifica di quelle esistenti

Ereditarietà

In definitiva, l'ereditarietà offre il vantaggio di ridurre i tempi di sviluppo, in quanto minimizza la quantità di codice da scrivere quando occorre:

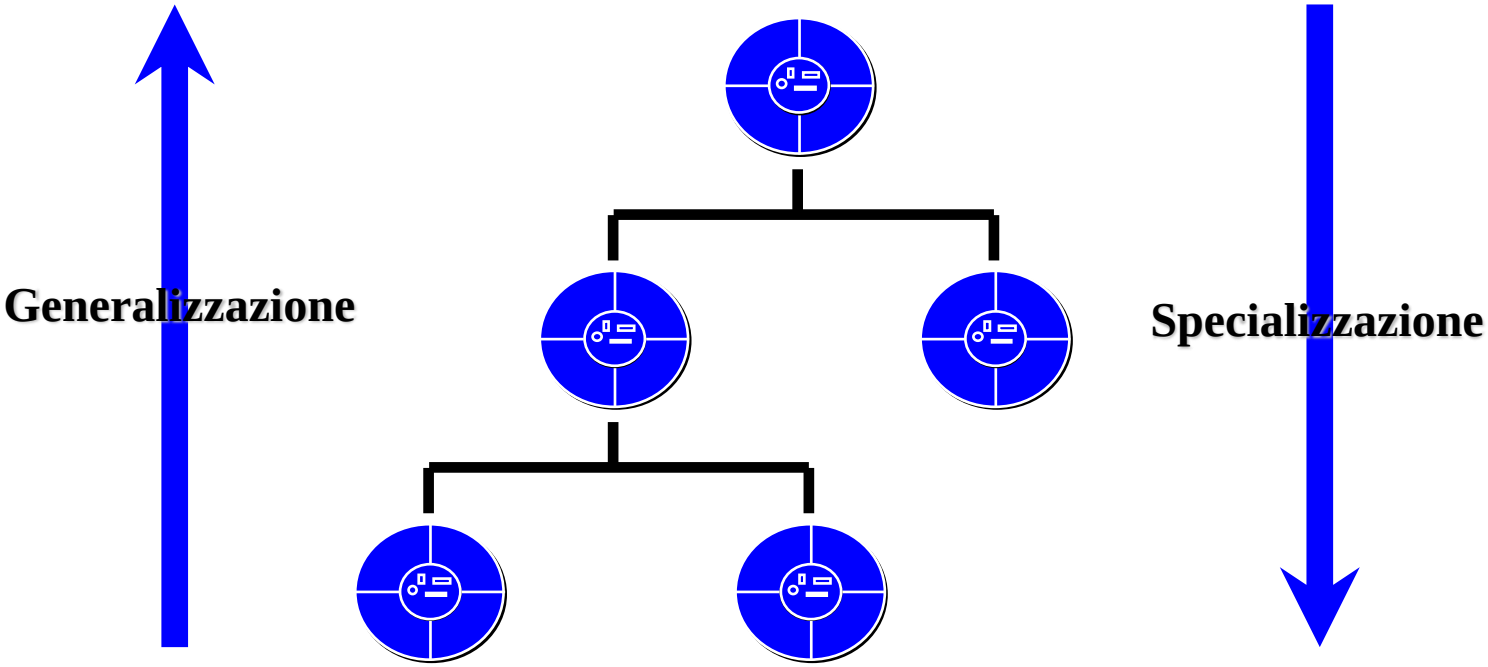
- definire un nuovo tipo d'utente che è un sottotipo di un tipo già disponibile, oppure
- adattare una classe esistente alle proprie esigenze

Non è necessario conoscere in dettaglio il funzionamento del codice da riutilizzare, ma è sufficiente modificare (mediante aggiunta o specializzazione) la parte di interesse

Ereditarietà

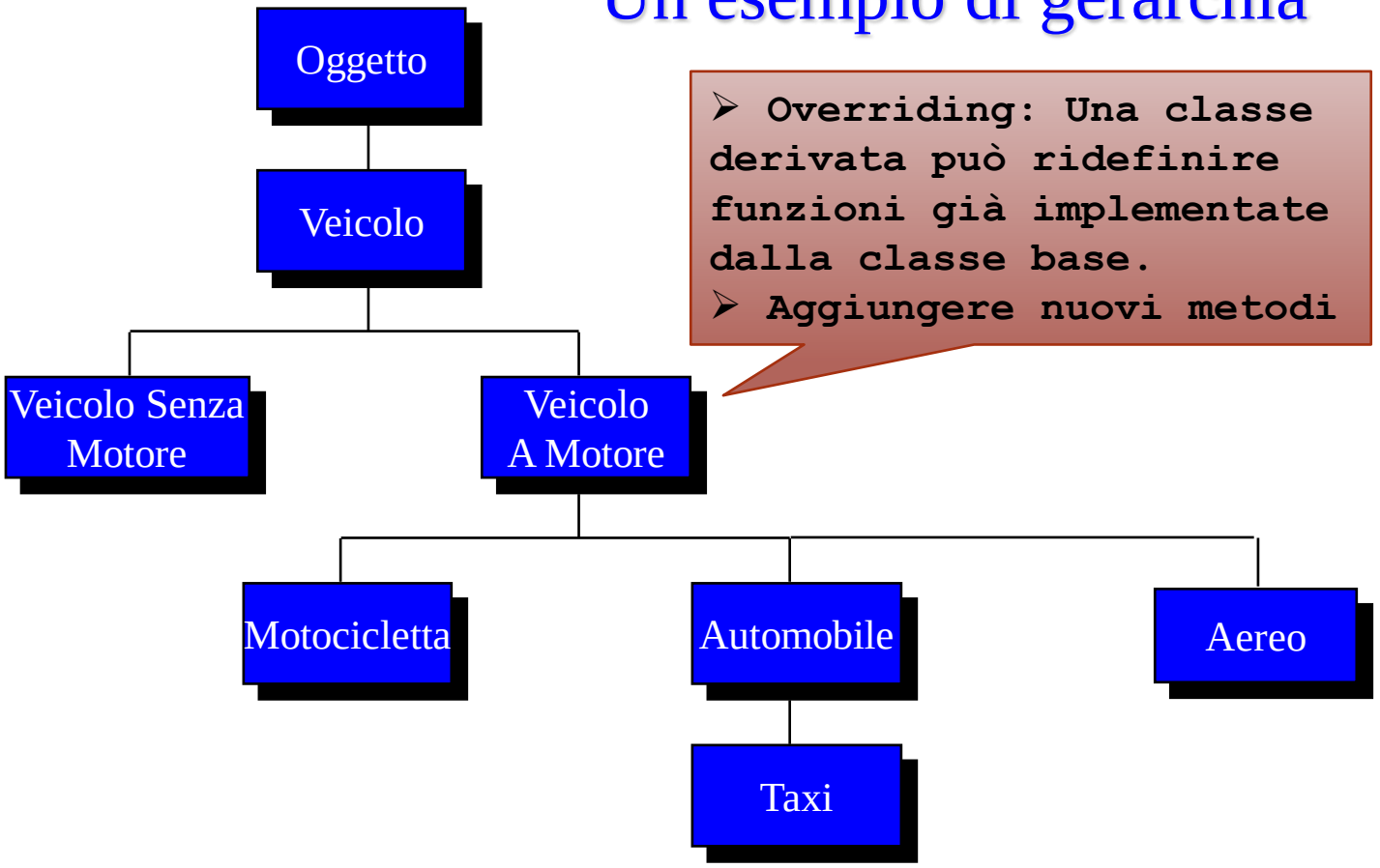
Generalizzazione: dal particolare al generale

Specializzazione o particolarizzazione: dal particolare al generale



Ereditarietà

Un esempio di gerarchia



- **Overriding: Una classe derivata può ridefinire funzioni già implementate dalla classe base.**
- **Aggiungere nuovi metodi**

Ereditarietà

L' ereditarietà è una tecnica che permette di descrivere una nuova classe, nei termini di un'altra già esistente

Essa consente inoltre di compiere le modifiche o estensioni necessarie.

La sottoclasse eredita tutti i metodi (operazioni) della genitrice, e può essere estesa con metodi ed attributi locali che ne completano la descrizione (**specializzazione** e **generalizzazione**)

Tale meccanismo consente di derivare una sottoclasse da una classe data per aggiunta, per occultamento o per ridefinizione di uno o più membri rispetto alla classe di partenza (che diventa una **superclasse** della nuova classe)

Specializzazione della classe derivata

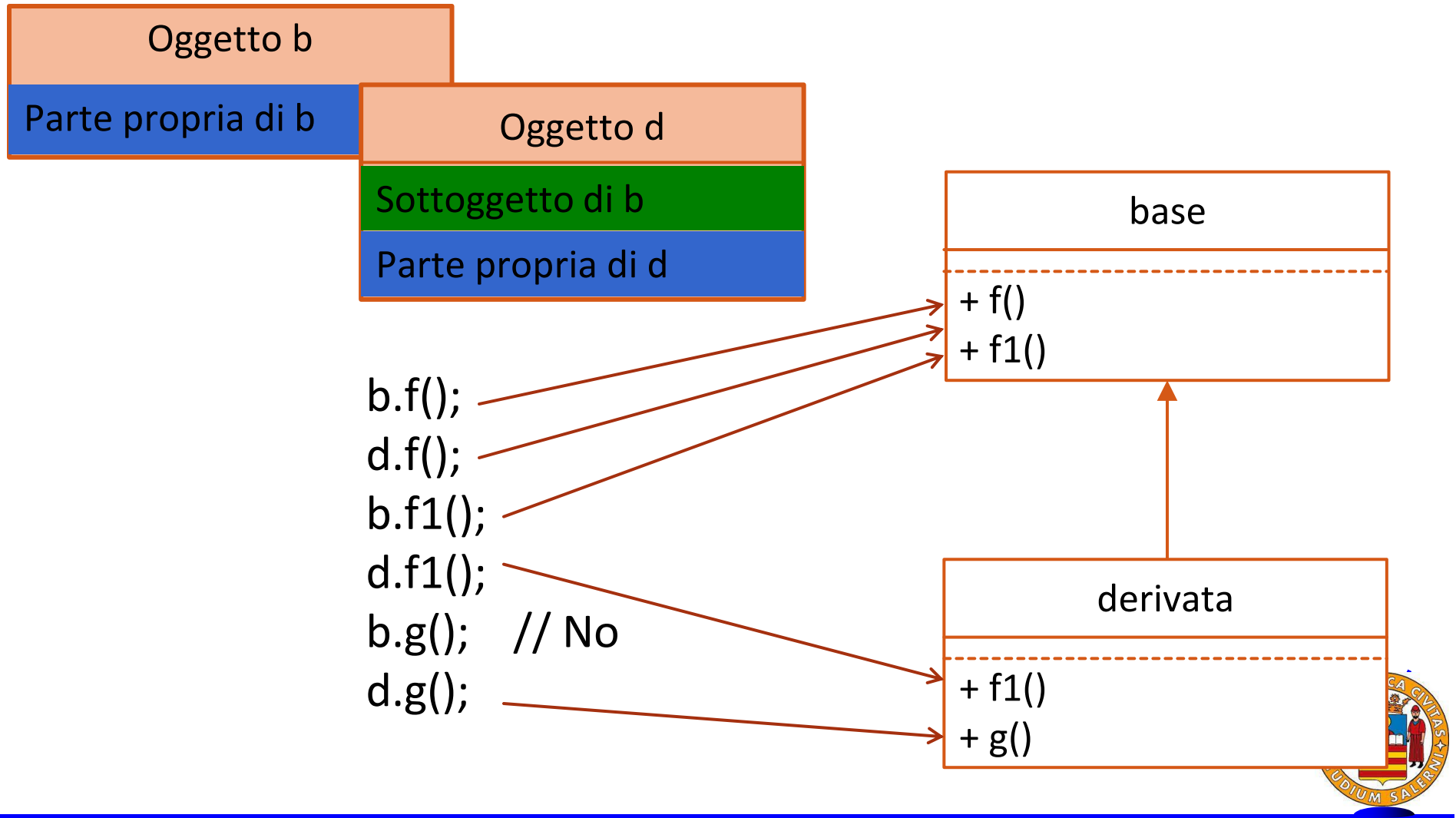
La classe ereditata è detta **classe base**

La classe che eredita è detta **classe derivata**

La classe che eredità può:

- Utilizzare i metodi della classe ereditata
- Modificare i metodi della classe ereditata
- Definire nuovi metodi propri

Esempio


Oggetto b
Parte propria di b
Oggetto d
Sottoggetto di b
Parte propria di d
base
+ f()
+ f1()
b.f();
d.f();
b.f1();
d.f1();
b.g(); // No
d.g();
derivata
+ f1()
+ g()

Meccanismi dell'ereditarietà

'Extends' (java), **':'** (C++) indicano quale classe si vuole ereditare

- JAVA permette di ereditare una sola classe
- C++ supporta l'ereditarietà multipla

Overriding *(dare precedenza)*

- Si ridefinisce il metodo della classe base specializzandone o sostituendone il comportamento (Esempio pulisci)

Overloading

- In Java l'overload di un metodo della classe base, effettuato nella classe derivata, non causa oscuramento

Shadowing

- Oscurare un metodo o un attributo della classe madre dichiarandolo privato

Ereditarietà nel Linguaggio Java

Sintassi:

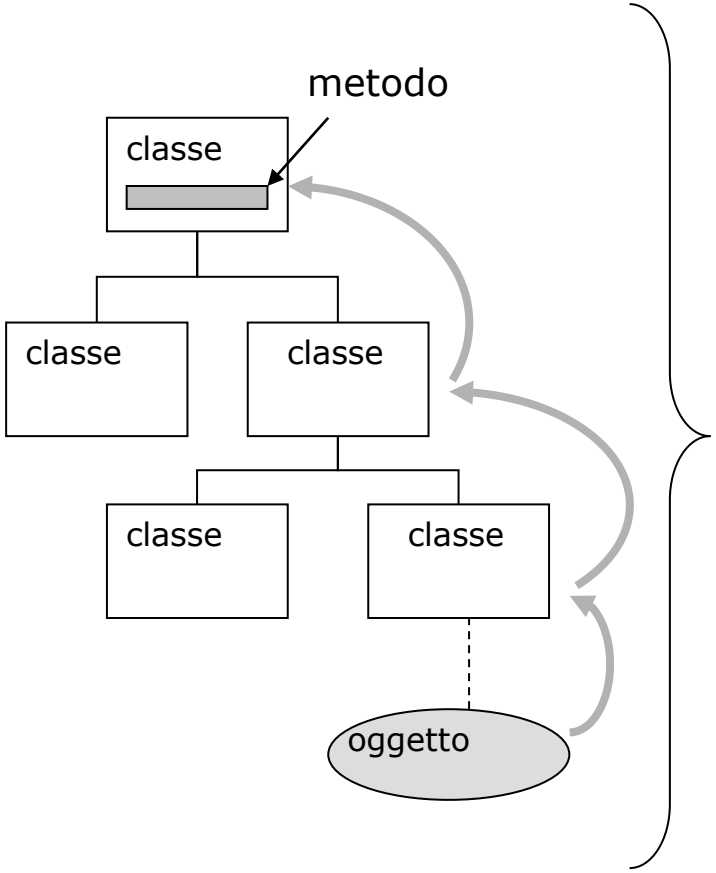
class Veicolo **:** public Oggetto {…};

class Veicolo extends Oggetto {…};

Il legame gen-spec tra due classi è rappresentato per mezzo della parola chiave extends, secondo la seguente sintassi:
{nome_classe_derivata} extends {nome_classe_base}

- In Java, a differenza del C++, non è possibile esprimere la modalità di derivazione (public, protected, private), che è sempre pubblica.

Ereditarietà nel Linguaggio Java



Quando si richiama un metodo su un oggetto, l'interprete ne cerca dapprima la definizione nella classe dell'oggetto stesso; se non la trova, cerca nella superclasse e risale la gerarchia fino a trovarla.

Nel caso esistano più metodi con lo stesso nome, tipo restituito e stessi parametri (firma) viene eseguito il metodo trovato per primo.

Esempio

```
class Base {
  public void stampa (int i) {
          System.out.println("Stampa"+i);}
}

public class Derivata extends Base {
  public void leggi(int i) {
          System.ou.println("Leggi"+i);}

public stati void main(String args[]){
      Derivata f=new Derivata();
      f.leggi(3);
      f.stampa(5);
   }
}
```

Overloading: Esempio

```
class BaseNonno{
  public void stampa ()
      { System.out.println("stampa()");} }

class BasePadre extends BaseNonno{
  public void stampa (int i)
      { System.out.println("stampa(int)");} }

public class Derivata extends BasePadre{
  public void stampa (float i)
      {   System.ou.println("stampa(float)");}

public stati void main(String args[]){
       Derivata f=new Derivata();
          f.stampa(5); f.stampa(3.5); f.stampa(); }
}
```

Esempio: classe base

```
class Struccante {
  private String s = new String(“Detersivo");
  public String toString() { return s; }

  public void append(String a) { s += a; }

  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void pulisci() { append(" pulisci()"); }

  public static void main(String[] args) {
      Struccante x = new Struccante();
      x.dilute(); x.apply(); x.pulisci();
      System.out.println(x.toString());
  }
}
```

Esempio: classe derivata

```
public class Detergente extends Struccante {
    // Occultare all'esterno
    private void dilute() {}

    // Specializzazione: overriding
    public void pulisci() { append(" Detergente.pulisci()"); }

    //aggiunta di un nuovo metodo
    public void spuma() { append(" spuma()"); }

    public void pulisci-old() {
                // Chiamata del metodo della classe ba
                super.pulisci(); //non si tratta di una ricorsione!!
                System.out.println("Puluto");}

   public static void main(String[] args) {
                        Detergente x = new Detergente();
                        x.apply(); x.spuma();
                         x.dilute();  // Solo allinterno della classe detergente
                         x.pulisci(); x.pulisci-old();
                        System.out.println(x.toString());
}} ///:~
```

NB: super punta alla classe base

Modalità di Protezione

protected

- Nuovo specificatore di accesso. Indica che l'accesso a quell'attributo è consentito a tutte le classi del package ed a quelle figlie.

Riepilogo Visibilità nel Linguaggio Java

I seguenti specificatori indicano quando l'attributo/metodo a cui si riferiscono è utilizzabile e da chi:

Accesso privato	si può accedere solo dall'ambito della stessa classe
Accesso di default	… dello stesso package
Accesso protetto	… delle sottoclassi e dello stesso package
Accesso pubblico	Completamente disponibile per qualsiasi altra classe che voglia farne uso

Ereditarietà e Costruttori

- Quando si crea un oggetto della classe derivata, questo contiene al suo interno un sotto-oggetto della classe base.
- Il sotto-oggetto va opportunamente inizializzato per mezzo della chiamata al costruttore (super(..)); Tale chiamata può essere:
  - Implicita: se la classe base ha un costruttore a zero argomenti.
  - Esplicita: se il costruttore della classe base prevede almeno un argomento.

- Quando non viene esplicitata l'invocazione al costruttore della classe base, il compilatore java automaticamente lo pone in testa al costruttore della classe derivata.

Esempio

```
class Art {
    Art() {System.out.println("Art constructor");}
}

class Drawing extends Art {
    Drawing() {System.out.println("Drawing constructor");}
}

public class Cartoon extends Drawing {
    public Cartoon()
            {System.out.println("Cartoon constructor");}

    public static void main(String[] args)
            {Cartoon x = new Cartoon();}
}///:~
```

Output

Art constructor

Drawing constructor

Cartoon constructor

Costruttori con argomenti

Una classe base può presentare diversi costruttori che si distinguono per numero, tipo o ordine dei parametri

In tal caso è il programmatore che deve scegliere quale costrutto della classe base chiamare

Si utilizza la parola chiave **super** per richiamare il costruttore desiderato

Esempio

```
class Game {
      Game(int i) { System.out.println("Game constructor“+i);}
      Game(flot j) { System.out.println("Game constructor“+j);}
}

class BoardGame extends Game {
  BoardGame(int i) {
     super(i);
     System.out.println("BoardGame constructor“+i);
}}

public class Chess extends BoardGame {
   Chess() {
      super(11);
      System.out.println("Chess constructor“);
    }

    public static void main(String[] args) {Chess x = new Chess();}
} ///:~
```

Sviluppo incrementale

Occorre notare che:

- È possibile sviluppare un sistema di oggetti in modo incrementale
- Ogni nuova classe completa quella ereditata aggiungendo delle caratteristiche e senza modificare il codice della classe base
- Se la classe ereditata funzionava correttamente siamo sicuri che un errore non può che trovarsi nel nuovo codice
- Siamo sicuri che un errore non pregiudica il funzionamento della classe ereditata

Considerazoni

Ereditarietà: Costruttori e distruttori

I costruttori sono metodi speciali

Costruzioni di classi derivate:

- Viene chiamato ricorsivamente il costruttore della classe base
- Si comincia la costruzione dalla radice della gerarchia
- Si istanziano gli oggetti creati nella parte dichiarativa
- Si eseguono i costruttori

La distruzione avviene al contrario

- Occorre distruggere prima gli elementi della classe derivata

Esempio

```
class Meal { Meal() { System.out.println("Meal()"); } }
class Bread { Bread() { System.out.println("Bread()"); } }
class Cheese { Cheese() { System.out.println("Cheese()"); } }
class Lettuce { Lettuce() { System.out.println("Lettuce()"); } }

class Lunch extends Meal { Lunch() { System.out.println("Lunch()"); } }

class PortableLunch extends Lunch {
    PortableLunch() { System.out.println("PortableLunch()");} }

public class Sandwich extends PortableLunch {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();

  public Sandwich() { System.out.println("Sandwich()"); }

  public static void main(String[] args) {new Sandwich();}
} ///:~
```

Output

"Meal()"

"Lunch()"

"PortableLunch()"

"Bread()“

"Cheese()"

"Lettuce()"

"Sandwich()"

La classe java.lang.Object

In Java:

- Gerarchia di ereditarietà semplice
- Ogni classe ha una sola super-classe

Se non viene definita esplicitamente una super-classe, il compilatore usa la classe predefinita **Object**

- Object non ha super-classe!

Metodi di Object

Object definisce un certo numero di **metodi pubblici**

- Qualunque oggetto di qualsiasi classe li eredita
- La loro implementazione base è spesso minimale
- La tecnica del polimorfismo permette di ridefinirli

public boolean **equals**(Object o)

- Restituisce "vero" se l'oggetto confrontato è identico (ha lo stesso riferimento) a quello su cui viene invocato il metodo
- Per funzionare correttamente, ogni sottoclasse deve fornire la propria implementazione polimorfica

Metodi di Object

public String **toString**()

- Restituisce una rappresentazione stampabile dell'oggetto
- L'implementazione base fornita indica il nome della classe seguita dal riferimento relativo all'oggetto (java.lang.Object@10878cd)

public int **hashCode**()

- Restituisce un valore intero legato al contenuto dell'oggetto
- Se i dati nell'oggetto cambiano, deve restituire un valore differente
- Oggetti "uguali" **devono** restituire lo stesso valore, oggetti diversi **possono** restituire valori diversi
- Utilizzato per realizzare tabelle hash

Controllo Override

- Pertanto molti linguaggi supportano opportuni controlli introducendo nuove parole chiavi. La soluzione di Java è quello di usare un'annotazione:

```
public class Animale{
  private String nome;

  @Override
  public boolean equals(Animale animale){
    return nome.equals(animale.nome);
  }
}
```

- In questo caso il compilatore ci restituisce un errore:

  The method equals(Animale) of type Animale must override or implement a supertype method

Controllo Override

- Ecco la soluzione corretta:

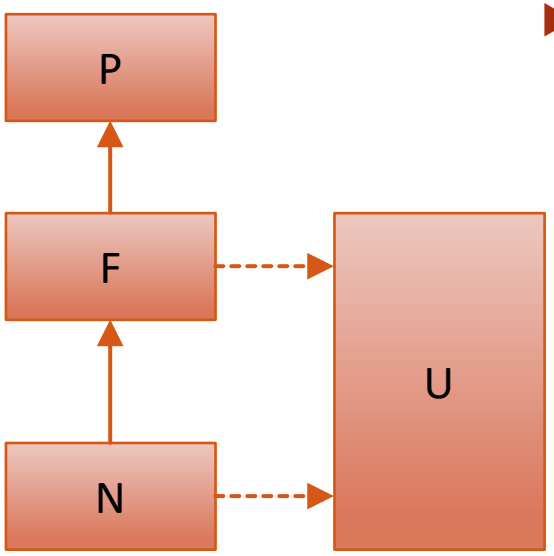
```
public class Animale{
  private String nome;

  @Override
  public boolean equals(Object animale){
    return nome.equals(((Animale) animale).nome);
  }
}
```

- Come evitare che un metodo di una classe base possa essere sovrascritto in quella derivata?

Cosa accade in C++

Modalità di Protezione

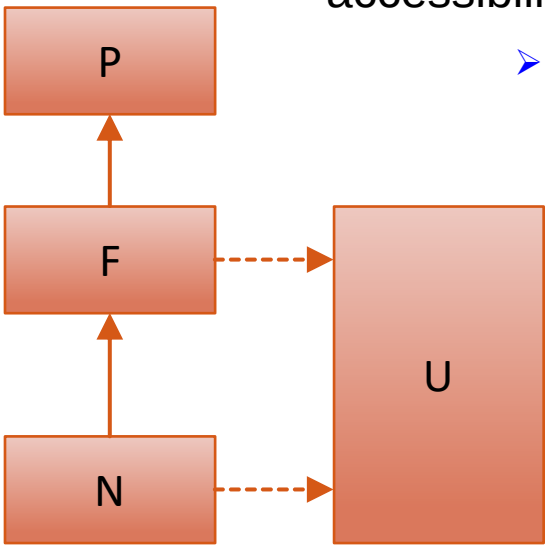
- Consideriamo la seguente gerarchia di classi:
  - P: classe base della gerarchia di derivazione;
  - F: sotto-classe derivata direttamente da P;
  - N: sotto-classe, derivata da F, ed indirettamente da P;
  - U: classe o funzione utente della gerarchia di derivazione.



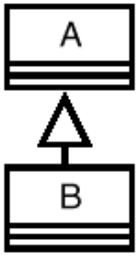
- Esistono tre modalità di protezione di membri:
  - **membri privati**: non accessibili all'esterno della classe che li ha definiti;
  - **membri pubblici**: accessibili dall'esterno della classe, siano esse classi derivate che utilizzatori
  - **membri protetti**: non accessibili all'esterno della classe dagli utilizzatori, ma accessibili da classi derivate.
- Sono definiti dal progettista di una classe e non possono essere modificati con la derivazione.

*Modalità di Protezione

- In C++ esistono tre modalità di derivazione, non disciplinano i diritti di accesso ai membri della classe da cui si deriva, ma come tali diritti sono trasmessi:
  - se F deriva da P in modalità pubblica – i diritti di accesso di F a P sono trasmessi a N e U, ovvero la sua parte pubblica è accessibile da N e U, mentre la parte protetta solo a N;
    - se si deriva F da P in modalità protetta – i diritti di accesso di F a P sono trasmessi a N e non a U, la sua parte pubblica e protetta sono accessibili da N mentre U non accede a nessuna parte di P;
      - se si deriva F da P in modalità privata (modalità di default) – i diritti di accesso di F a P non sono trasmessi.



membri in A	*membri in B, se B eredita da A in maniera ...*		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	*non accessibile*	*non accessibile*	*non accessibile*



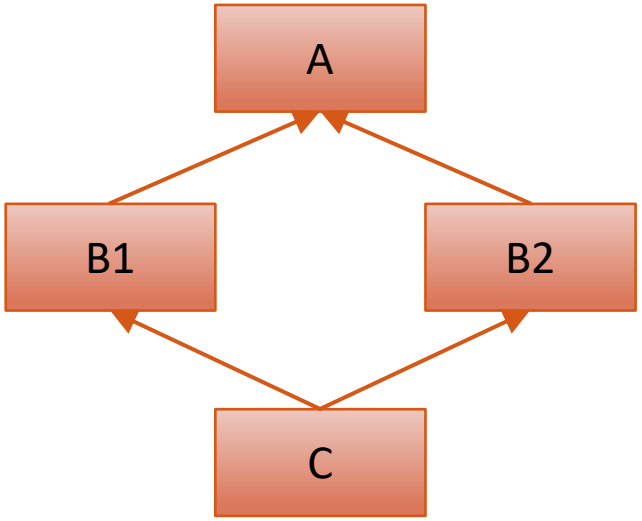
Cardinalità

- In c++, supponiamo che una certa classe C derivi, per eredità multipla, da due classi genitrici B1 e B2. Nella definizione di C, il nome di ognuna delle due classi base deve essere preceduto dal rispettivo specificatore di accesso (se non è private, che, ricordiamo, è lo specificatore di default in C++). Per esempio:

```
class C : protected B1, public B2 { ..... };
```

- in questo caso, nella classe C, i membri ereditati da B1 sono tutti protetti, mentre quelli ereditati da B2 rimangono come erano nella classe base (protetti o pubblici).
- Supponiamo ora che le classi B1 e B2 derivino a loro volta da un'unica classe base A. Per cui, quando viene istanziata C, sono costruite direttamente soltanto le sue dirette genitrici B1 e B2, ma ciascuna di queste costruisce separatamente A.

Cardinalità



- La replicazione di una classe base può causare problemi:
  - occupazione doppia di memoria, che può essere poco "piacevole", soprattutto se gli oggetti di C sono molti e il sizeof(A) è grande;
  - errore di ambiguità: se gli oggetti di C accedono direttamente ai membri ereditati da A, il compilatore darebbe errore, non sapendo se accedere ai membri ereditati tramite B1 o tramite B2.

Cardinalità

- Il secondo problema può essere qualificando ogni volta i membri ereditati da A:

```
ogg.B1::ma      // indica che ma è ereditato tramite B1
ogg.B2::ma      // indica che ma è ereditato tramite B2
```

Cardinalità

- Entrambi i problemi, invece, si possono risolvere in C++ definendo A come classe base "virtuale", inserendo, nelle definizioni di tutte le classi derivate, la parola-chiave virtual accanto allo specificatore di accesso alla classe base. Esempio:

  ```
  class B1 : virtual protected A  {  ……..  } ;
  class B2 : virtual public A  {  ……..  } ;
  ```

- La parola-chiave virtual non ha alcun effetto sulle istanze dirette di B1 e di B2: ciascuna di esse costruisce la propria classe base normalmente, come se virtual non fosse specificata.
- Se viene istanziata la classe C, derivata da B1 e da B2 per eredità multipla, viene creata una sola copia dei membri ereditati da A, della cui inizializzazione deve essere lo stesso costruttore di C ad occuparsene (contravvenendo alla regola generale che vuole che ogni figlia si occupi solo delle sue immediate genitrici).
- In altre parole, nella lista di inizializzazione del costruttore di C devono essere incluse le chiamate, non solo dei costruttori di B1 e di B2, ma anche del costruttore di A. In sostanza la parola-chiave virtual dice a B1 e B2 di non prendersi cura di A quando viene creato un oggetto di C, perchè sarà la stessa classe "nipote" C ad occuparsi della sua "nonna".