TECNOLOGIA OBJECT-ORIENTED

Il software dovrebbe essere **protetto, riusabile, documentato, modulare, incrementalmente estendibile**

L'OBIETTIVO
 costruzione ***modulare*** **e** ***incrementale*** del software ottenuta per **estensione** / **specializzazione** di ***componenti***

- tecnologia ***component-based***

LE NECESSITÀ
 servono strumenti per la costruzione ***modulare*** **e** ***incrementale*** del software

UN MECCANISMO RISOLUTIVO: EREDITARIETÀ
 l'ereditarietà serve per *non ripartire da zero* quando serve una nuova classe, permettendo di ***poterla definire a partire da una già esistente*** nella logica del ***riuso***

Bisognerà specificare ciò che la distingue:
 cosa la nuova classe ha *in più* rispetto alla precedente sia in termini di *dati*, sia in termini di *operazioni*.
 cosa la nuova classe ha di *diverso* rispetto alla precedente sia in termini di *operazioni*.

- ogni classe derivata ***eredita*** tutte le proprietà della classe base di partenza.

LA CLASSE DERIVATA PUÒ:
 aggiungere **nuovi campi dati e nuovi metodi** a quelli ereditati dalla classe-base
 ridefinire **alcuni dei metodi** ereditati dalla classe-base

LA CLASSE DERIVATA NON PUÒ:
 eliminare campi dati o metodi (comportamento monotono, ***si accresce sempre***)

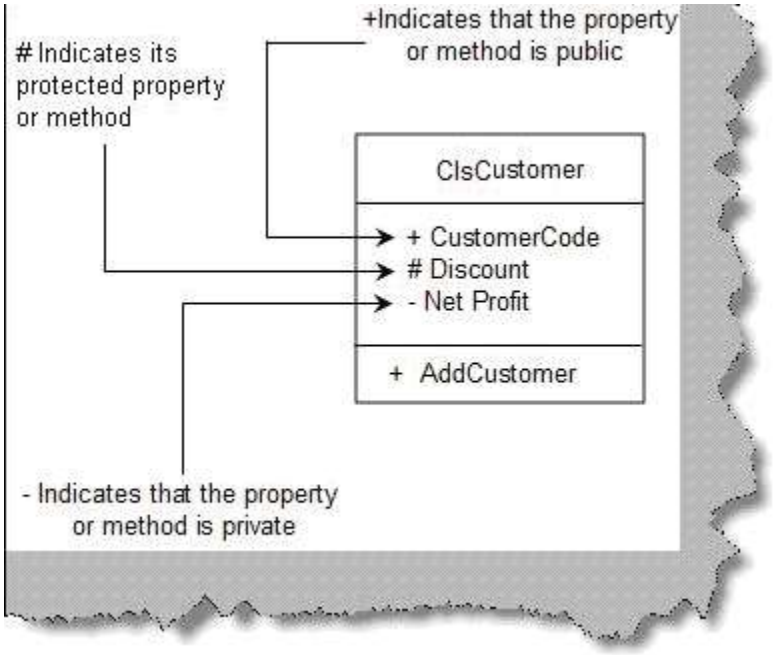
VISIBILITÀ

Nella classe-base:
 ciò che è *privato* è visibile *solo ai metodi della classe stessa*
 ciò che è *di package* è visibile *solo ai metodi delle classi dello stesso package*
 ciò che è *pubblico* è *visibile a tutti*

Perciò, la classe derivata:
 non può vedere la parte privata *della classe-base****,*** appunto perché privata;
 può vedere la parte di package *della classe-base****,*** purché la nuova classe sia definita nello stesso package
 vede, ovviamente, la parte pubblica *della classe-base****,*** in quanto visibile a tutti.

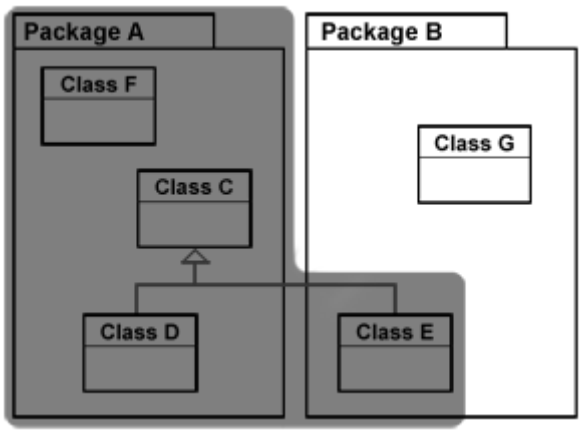
Ma non basta…..

Per sfruttare davvero l'ereditarietà occorre ***superare la rigidità*** dei tre livelli di visibilità privato / pubblico / package: ***occorre una visibilità specifica per le classi derivate*** → **`protected`**



Un campo dati o un metodo `protected`:

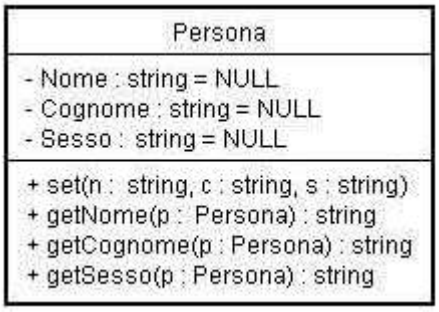
- **è visibile alle classi derivate *indipendentemente dal package* in cui esse sono definite**
- **per il resto si comporta come la visibilità *package*.**



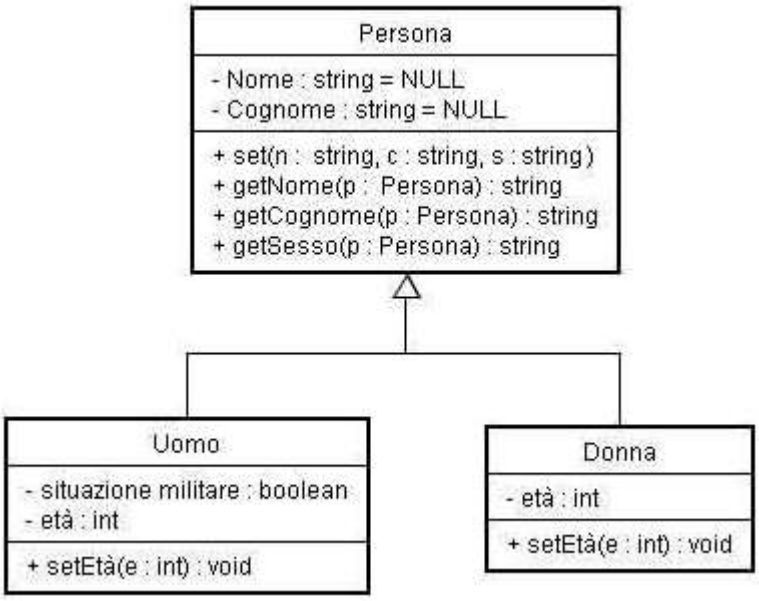
La qualifica `protected`: occorre dunque valutare con attenzione quando sia opportuna.

- rende visibile un attributo *a tutte le sottoclassi* di quella classe, *presenti e future*
- perciò, costituisce un *permesso di accesso "indiscriminato"*, valido per *ogni possibile sottoclasse che possa in futuro essere definita*, senza possibilità di distinzione.

Esemplifichiamo con **UML** la classe Persona:



ed ampliamo l'esempio precedente della classe Persona creando due ***sottoclassi*** Uomo e Donna:



Con il meccanismo della ereditarietá si ereditano:

- *tutti i campi dati* (anche quelli *private* a cui la classe derivata **non potrà accedere direttamente**)
- *tutti i metodi* (anche quelli *private* che la classe derivata **non potrà usare direttamente**)

tranne i costruttori

- I costruttori, infatti, non svolgono *funzioni*, ma inizializzano un'istanza della classe

  ✎ sono **specifici di quella particolare classe**

Quindi…..

Per costruire un oggetto di una classe derivata, è *sempre necessario* chiamare un costruttore della classe-base, in quanto:

- solo il costruttore della classe base può sapere come inizializzare i campi-dati ereditati da tale classe in modo corretto

- è il solo modo per garantire l'inizializzazione di campi-dati privati (a cui la sottoclasse non può accedere direttamente)

- si evita una inutile duplicazione di codice nella sottoclasse.

Q*uale costruttore della classe-base* viene chiamato?

 il costruttore di *default* definito da Java (se noi non lo definiamo) chiama il costruttore di *default* della classe base (se questo non esiste si ha errore). Tale costruttore è *pubblico*, in modo da consentire alla classe di essere normalmente istanziata. Tale costruttore o non fa niente o, se la classe è derivata, invoca il costruttore della superclasse

 per i costruttori definiti dall'utente occorre *specificare esplicitamente* quale costruttore della classe base vada chiamato, mediante la notazione **`super(..)`**, *che dev'essere la prima istruzione del corpo del costruttore.* Se questa manca, Java inserisce automaticamente una chiamata al costruttore di default[1] della classe base (`super()`)

```
class NomeFiglia {

        public NomeFiglia (){

                super();
        }
 }
```

EREDITARIETÀ e OMONIMIE

NB: `this` è un riferimento all'istanza corrente
 `super` è un riferimento alla classe base

Sono espressioni lecite:

- `this.val` che indica il campo `val` della classe dell'istanza corrente
- `this.f()` che richiama il metodo `f()` della classe dell'istanza corrente
- `super(..)` che **richiama un costruttore della classe base**
- `super.val` che indica il campo `val` della classe base
- `super.f()` che richiama il metodo `f()` della classe base

Le ultime due notazioni sono utili se la sottoclasse definisce un campo dati o un metodo ***omonimo*** con uno della classe base.

POLIMORFISMO

*Una funzione si dice **POLIMORFA** se è capace di operare su oggetti di tipo diverso **specializzando il suo comportamento in base al tipo dell'oggetto su cui opera***

I linguaggi a oggetti hanno intrinseco il polimorfismo, in quanto possono esistere – in classi diverse – funzioni con lo stesso nome ma con effetti completamente diversi: operazioni di ***overriding*** (***ridefinizione*** di metodi in sottoclasse) oppure di ***overloading*** (definizione di metodi con lo stesso nome ma differente numero e/o tipo di parametri, ne sono tipico esempio i diversi costruttori di una classe)

estratto da lucidi realizzati da Prof. Franco Zambonelli e Ing. Enrico Denti

[1] *A meno che* non ci sia già una chiamata a un altro costruttore della stessa classe mediante la notazione `this(..)`.