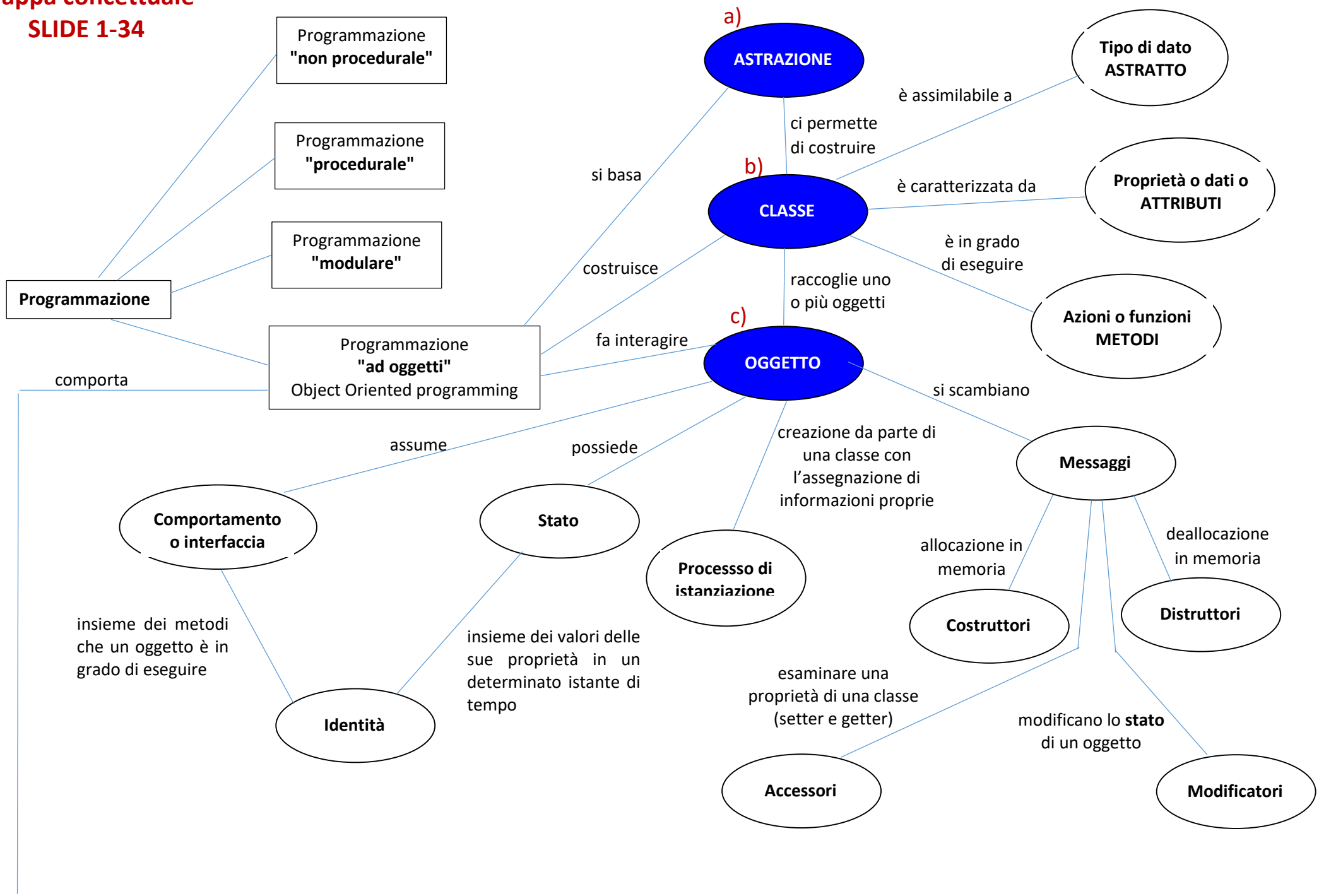
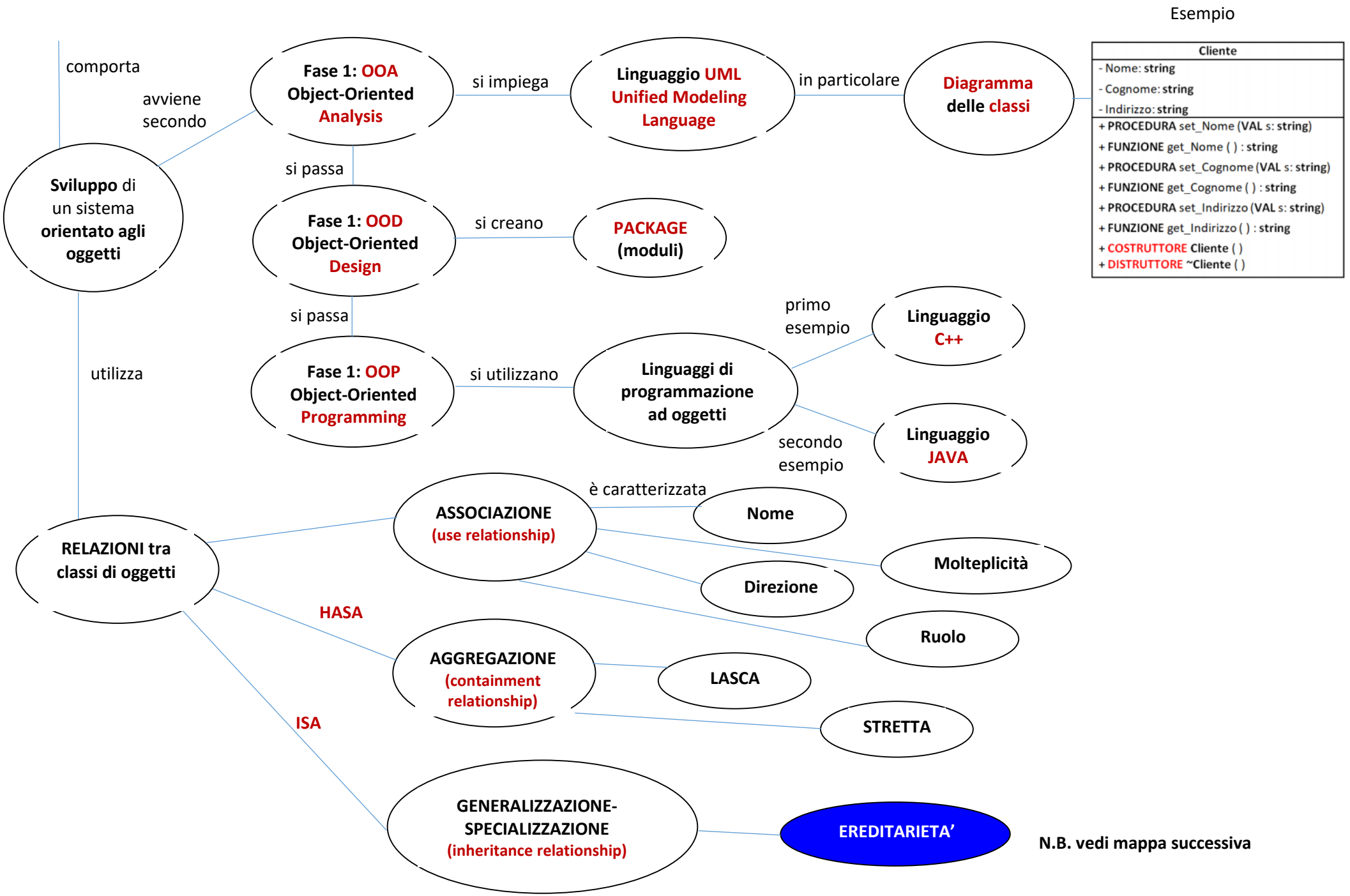
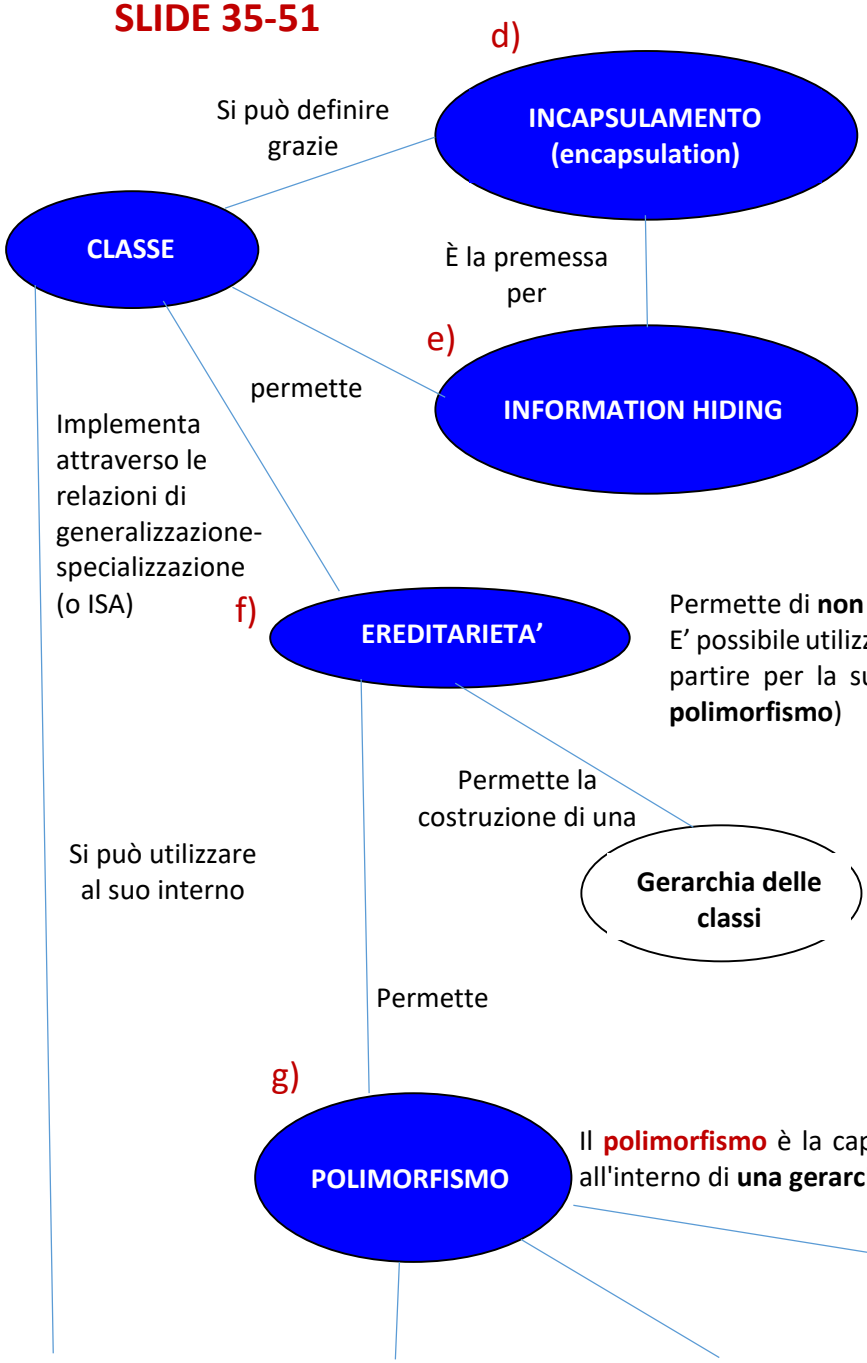
Mappa concettuale
SLIDE 1-34




Esempio
comporta
avviene secondo
Sviluppo di un sistema orientato agli oggetti
Fase 1: OOA Object-Oriented Analysis
si impiega
Linguaggio UML Unified Modeling Language
in particolare
Diagramma delle classi
Cliente
- Nome: string
- Cognome: string
- Indirizzo: string
+ PROCEDURA set_Nome (VAL s: string)
+ FUNZIONE get_Nome ( ) : string
+ PROCEDURA set_Cognome (VAL s: string)
+ FUNZIONE get_Cognome ( ) : string
+ PROCEDURA set_Indirizzo (VAL s: string)
+ FUNZIONE get_Indirizzo ( ) : string
+ COSTRUTTORE Cliente ( )
+ DISTRUTTORE ~Cliente ( )
si passa
Fase 1: OOD Object-Oriented Design
si creano
PACKAGE (moduli)
si passa
Fase 1: OOP Object-Oriented Programming
si utilizzano
Linguaggi di programmazione ad oggetti
primo esempio
Linguaggio C++
secondo esempio
Linguaggio JAVA
utilizza
RELAZIONI tra classi di oggetti
ASSOCIAZIONE (use relationship)
è caratterizzata
Nome
Molteplicità
Direzione
Ruolo
HASA
AGGREGAZIONE (containment relationship)
LASCA
STRETTA
ISA
GENERALIZZAZIONE-SPECIALIZZAZIONE (inheritance relationship)
EREDITARIETA'
N.B. vedi mappa successiva

Mappa concettuale
SLIDE 35-51



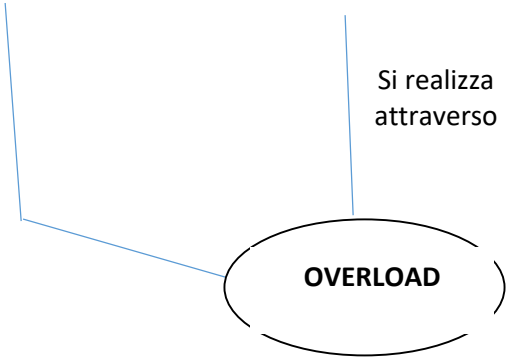
Indica la capacità di un oggetto di mantenere al suo interno gli attributi (o proprietà) e i metodi (funzioni) che agiscono su di essi (ossia stato e comportamento).
Si crea come una **capsula**, un contenitore concettuale, che isola l'oggetto dalle cose esterne.
Tutti gli attributi ed i metodi (le infotmazioni utili) sono ben localizzati (incapsulati) nell'oggetto

a) I dettagli implementativi di una classe possono essere nascosti all'utente (vedi clausole di visibilità **private** e **protected**) (**occultamento delle informazioni**)
b) Le scelte interne di progettazione e gli effetti che eventuali cambiamenti di tali decisioni comportano, sono nascoste ai possibili utilizzatori in quanto l'implementazione dei metodi – **public**, **private** o **protected** - non è comunque visibile all'esterno (**occultamento dell'implementazione**)

Permette di **non** creare una classe dal nulla.
E' possibile utilizzare una classe già esistente (**ereditarietà semplice**) o più di una (**ereditarietà multipla**) dalla quale partire per la sua creazione, specificando solo le differenze con quest'ultima(per **estensione** o attraverso il **polimorfismo**)

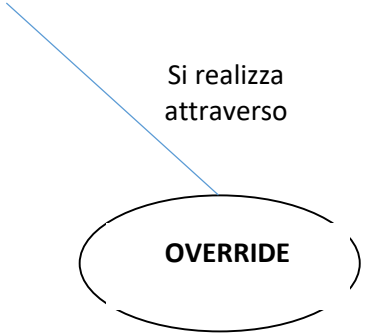
Una classe **eredita** dalle classi superiori nella gerarchia tutti i loro **metodi** e tutte le loro **proprietà** (ovviamente non definite private).
La classe diventa una **combinazione** dei **metodi** e delle **proprietà** di tutte le **superclassi** che la precedono nella gerarchia delle classi ai quali vanno aggiunti i metodi e le proprietà specifici di quella classe

Il **polimorfismo** è la capacità espressa dai metodi ridefiniti di assumere forme (ossia implementazioni) diverse all'interno di **una gerarchia di classi** o all'interno **di una stessa classe**



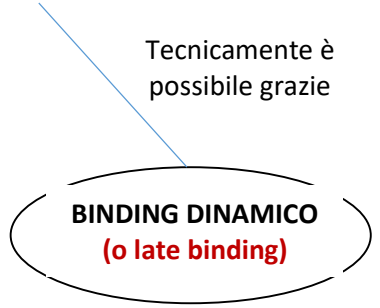
Con il termine **overload** si intende la scrittura di più metodi identificati dallo **stesso nome** che però hanno, in ingresso, **parametri di tipo e numero diverso**. Può avvenire **sovraccaricando** uno o più metodi della stessa classe oppure uno o più metodi ereditati da una superclasse.

OVERLOAD NON IMPLICA L' EREDITARIETA'



Con il termine **override** si intende una vera e propria **riscrittura** di un metodo all'interno di una classe mantenendo intatta la sua segnatura che abbiamo ereditato dalla superclasse.
Stessa segnatura significa che in entrambi i metodi sono perfettamente uguali il nome, l'eventuale tipo del valore di ritorno e numero e tipo dei parametri

OVERRIDE IMPLICA L' EREDITARIETA'



Il compilatore non genera una volta per tutte, all'atto della compilazione, il codice per l'assegnazione dei valori delle variabili in funzione delle chiamate dei metodi, o il codice per calcolare quale metodo chiamare in funzione delle informazioni provenienti dall'oggetto ma invece genera un codice che verrà utilizzato **per calcolare quale metodo richiamare di volta in volta**