

Le tre regole fondamentali

**Java a tre (4) regole fondamentali che oggi andremo ad approfondire
nella loro applicazione pratica**

1- Incapsulamento

2- Ereditarietà

3- Polimorfismo

4- Astrazione

Incapsulamento

Il significato di Encapsulation è assicurarsi che i dati "sensibili" siano nascosti agli utenti, aiuta altre caratteristiche come l'oscuramento dei dati, l'isolamento e il disaccoppiamento funzionale.

Per poter raggiungere l'obiettivo dell'incapsulamento, dobbiamo:

- **dichiarare variabili/attributi di classe come private**
 - **fornire metodi get e set pubblici per accedere e aggiornare il valore di una private variabile**
-

Nei linguaggi di programmazione orientata agli oggetti, il termine *incapsulamento* può essere usato per riferirsi a due concetti, collegati tra loro ma distinti :

- **un meccanismo del linguaggio atto a limitare l'accesso diretto agli elementi dell'oggetto;**
- **un costrutto del linguaggio che favorisce l'integrazione dei metodi propri della classe all'interno della classe stessa.**

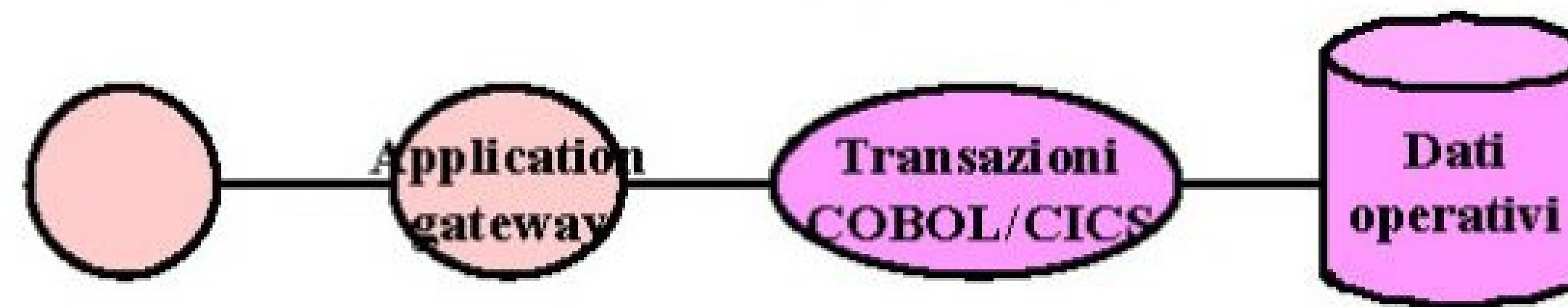
In generale si utilizza il primo significato da solo o in combinazione con il secondo come una funzionalità identificativa dei linguaggi di programmazione orientati agli oggetti, mentre altri linguaggi di programmazione che prevedono la chiusura, vedono l'incapsulamento come una funzionalità indipendente dall'orientamento agli oggetti.

La seconda definizione è motivata dal fatto che in diversi linguaggi di programmazione orientata agli oggetti l'occultamento degli elementi non è automatico o può essere scavalcato da altri modificatori di visibilità, pertanto l'occultamento delle informazioni è definito come concetto separato da chi lo preferisce alla prima.

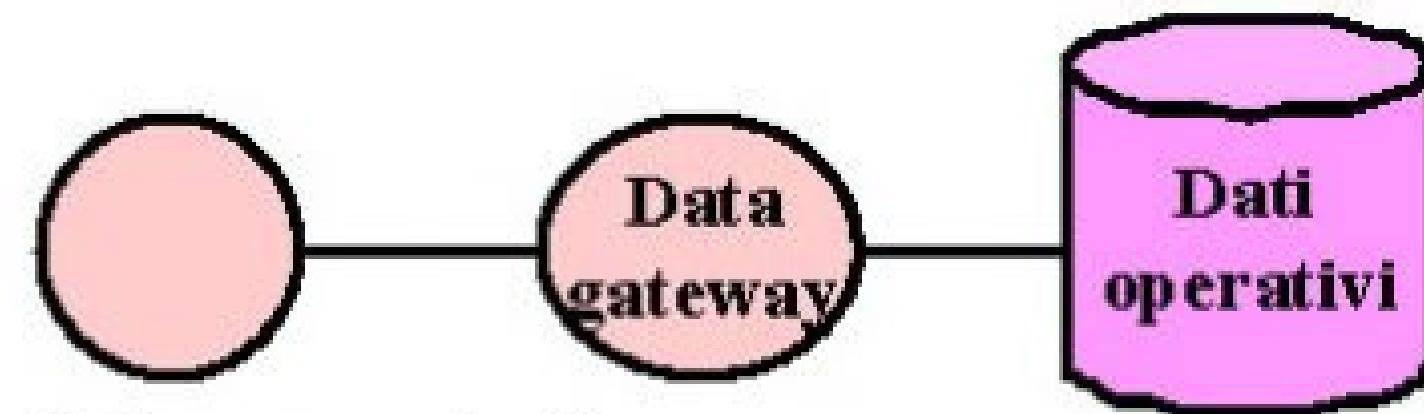
Cattura di
maschere video



Attivazione di
transazioni host



Accesso a dati host



Servizi esportati

Le parole chiave private/public/default è un modificatore di accesso , nel senso che viene utilizzata per impostare il livello di accesso per classi, attributi, metodi e costruttori.

Dividiamo i modificatori in due gruppi:

- **Modificatori di accesso : controlla il livello di accesso**
- **Modificatori non di accesso : non controllano il livello di accesso, ma forniscono altre funzionalità**

Per le classi possiamo usare:

- **Public:** La classe è accessibile da qualsiasi altra classe
- **Default:** La classe è accessibile solo dalle classi nello stesso pacchetto.
Questa viene usata quando non si specifica un modificatore.

Per attributi, metodi e costruttori , puoi utilizzare uno dei seguenti:

- **Public:** Il codice è accessibile per tutte le classi
- **Private:** Il codice è accessibile solo all'interno della classe dichiarata
- **Default:** Il codice è accessibile solo nello stesso pacchetto. Viene usato quando non si specifica un modificatore.
- **Protected:** I codice è accessibile nello stesso pacchetto e nelle stesse sottoclassi.

Per le classi , puoi utilizzare final o abstract:

- **final** La classe non può essere ereditata da altre classi
- **astratto** La classe non può essere usata per creare oggetti

Per attributi, metodi e costruttori , puoi utilizzare uno dei seguenti:

- **final:** Attributi e metodi non possono essere sovrascritti/modificati
 - **static:** Attributi e metodi che appartengono alla classe, piuttosto che ad un oggetto
 - **abstract:** può essere usato solo in una classe astratta, e può essere usato solo su metodi. Il metodo non ha un corpo. Il corpo è fornito dalla sottoclasse (ereditata da).
 - **synchronized:** Metodi accessibili solo da un thread alla volta
 - **volatile:** Il valore di un attributo non è memorizzato nella cache thread-local, ed è sempre letto dalla "memoria principale"
-

```
1. public class Main {
2.   final int x = 1020;
3.   final double PI = 3.14;
4.
5.   public static void main(String[] args) {
6.     Main myObj = new Main();
7.     myObj.x = 500; // genera un errore
8.     myObj.PI = 125; // genera un errore
9.     System.out.println(myObj.x);
10.  }
11.}
```

ESEMPIO

```
1. public class Main {  
2.     static void myStaticMethod() { // metodo statico  
3.         System.out.println("I metodi statici possono essere chiamati senza creare oggetti"); }  
4.  
5.     public void myPublicMethod() { // metodo pubblico  
6.         System.out.println("I metodi pubblici devono essere chiamati creando oggetti"); }  
7.  
8.     public static void main(String[ ] args) { // Main method  
9.         myStaticMethod(); // Chiamiamo lo static method  
10.        // myPublicMethod(); Questo produrrebbe un errore  
11.  
12.        Main myObj = new Main(); // Creiamo un oggetto main  
13.        myObj.myPublicMethod(); // Chiamo il metodo pubblico }
```

Con private quindi è possibile accedere alle variabili solo all'interno della stessa classe, una classe esterna non ha accesso ad esse.
Tuttavia, è possibile accedervi se forniamo metodi get e set pubblici.
Il metodo get restituisce il valore della variabile il metodo set ne imposta il valore. La sintass comunei è che iniziano con o get|set, seguito dal nome della variabile, con la prima lettera maiuscola:

```
public class Person {  
    private String name; // private = restricted access  
  
    public String getName() { // Getter  
        return name; }  
  
    public void setName(String newName) { // Setter  
        this.name = newName; }  
}
```

Get and Set

Proviamo ora in un'altra classe ad utilizzare i criteri precedentemente settati per creare l'oggetto:

```
public class Main {  
    public static void main(String[] args) {  
        Person myObj = new Person();  
        myObj.setName("Pippo"); // Settiamo il valore name = "pippo"  
        System.out.println(myObj.getName());  
    }  
}
```

Perché l'incapsulamento?

- **Migliore controllo degli attributi e dei metodi di classe**
 - **Gli attributi di classe possono essere resi di sola lettura (se usi solo il get come metodo) o di sola scrittura (se usi solo il set come metodo)**
 - **Flessibile: il programmatore può modificare una parte del codice senza influire sulle altre parti**
 - **Maggiore sicurezza e mascheramento dei dati**
-

Esercizio Incapsulamento

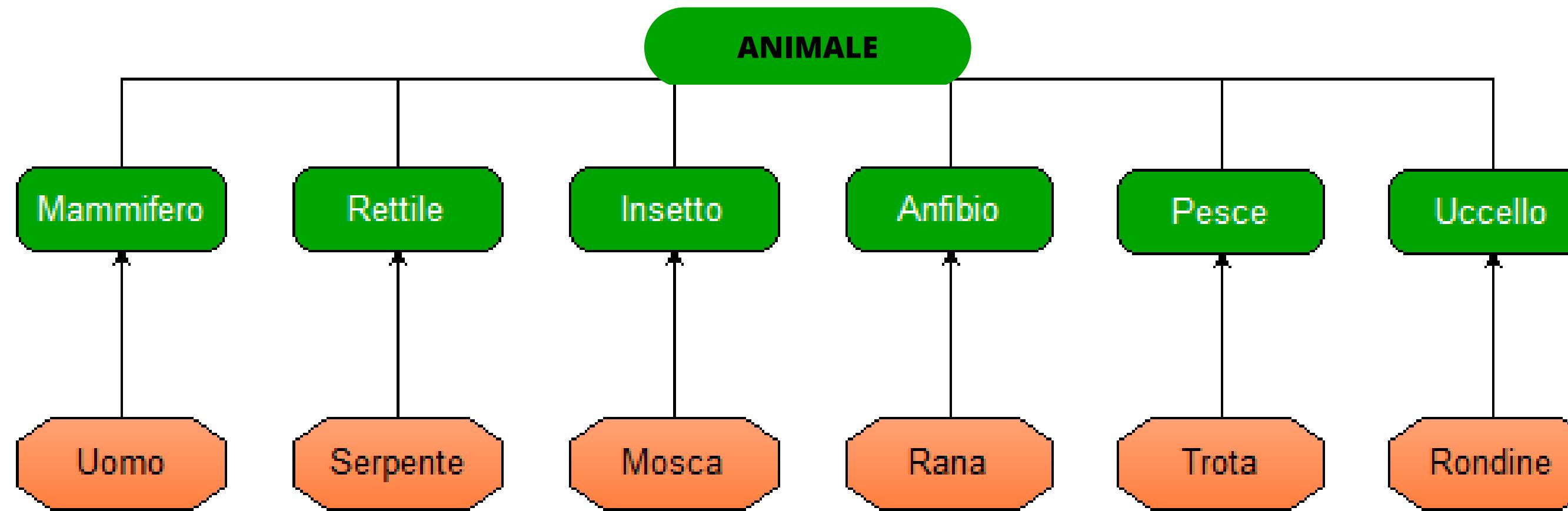
Creare una classe Piatto Speciale che ha ingredienti di tipo PRIVATE e ingredienti pubblic (es: un panino normale o di sesamo) e un prezzo specifico per ognuno di loro (es ingr1 = 2 euro) + una base di fisso (prezzo = 1 euro)

Creare poi una classe menu che permetta in loop di ordinare il piatto speciale selezionando tra una lista gli ingredienti public e facendo inserire a mano gli ingredienti PRIVATE dopo aver stampato una lista dei possibili casi (solo quelli devono essere accettati) a quel punto calcolare il totale e riportare all'inizio

EREDITARIETÀ

Il principio di ereditarietà si basa sul fatto di poter definire un legame di dipendenza di tipo gerarchico tra classi diverse. Una classe deriva da un'altra se da essa ne "eredita" il comportamento e le caratteristiche. La classe "figlia" si dice "classe derivata", mentre la classe "padre" prende il nome di "classe base" (superclasse). La classe base contiene il codice comune a tutte le classi da essa derivate.

Ogni classe derivata rappresenta invece una specializzazione della superclasse, ma eredita in ogni caso dalla classe base i comportamenti in essa definiti. Per fare un'esempio chiarificatore, supponiamo di voler rappresentare il genere animale definendo una struttura di classi equivalente. Nel seguente diagramma possiamo individuare una prima classe base "Animale" da cui tutte le altre derivano:



EREDITARIETÀ

In Java è possibile ereditare attributi e metodi da una classe all'altra. Raggruppiamo il "concetto di eredità" in due categorie:

- **sottoclasse (figlio)** - la classe che eredita da un'altra classe
- **superclasse (genitore)** - la classe da cui viene ereditata

Per ereditare da una classe, usa la `extends` parola chiave.

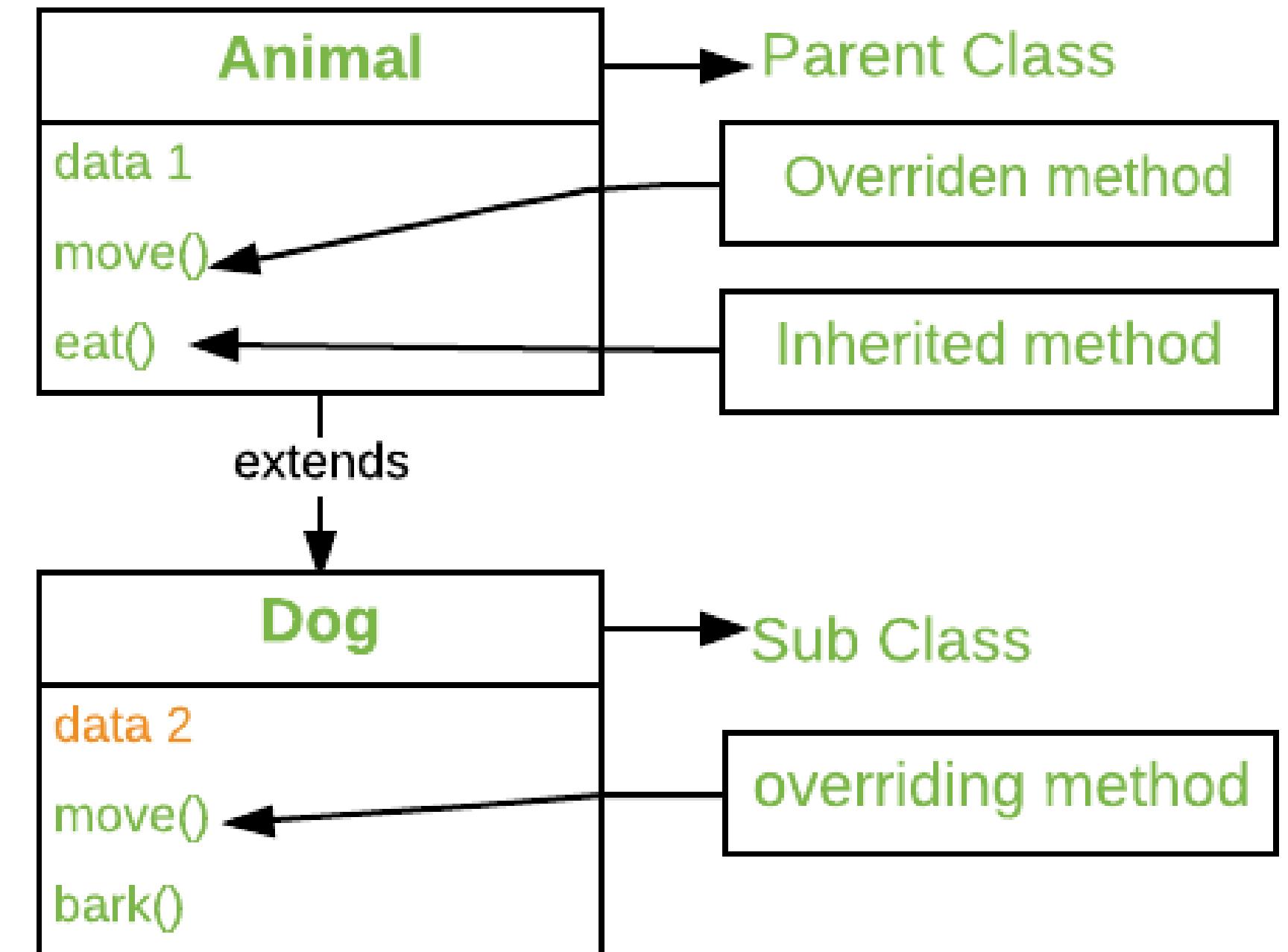
```
class Vehicle {  
    protected String brand = "Ford";  
    public void honk() {  
        System.out.println("Tuut, tuut!"); } }
```

```
class Car extends Vehicle {  
    private String modelName = "Fiat";  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.honk();  
        System.out.println(myCar.brand + " " + myCar.modelName);}  
}
```

OVERRIDE del metodo in Java

In qualsiasi linguaggio di programmazione orientato agli oggetti, Overriding è una funzionalità che consente a una sottoclasse o classe figlia di fornire un'implementazione specifica di un metodo che è già fornito da una delle sue superclassi o classi genitore.

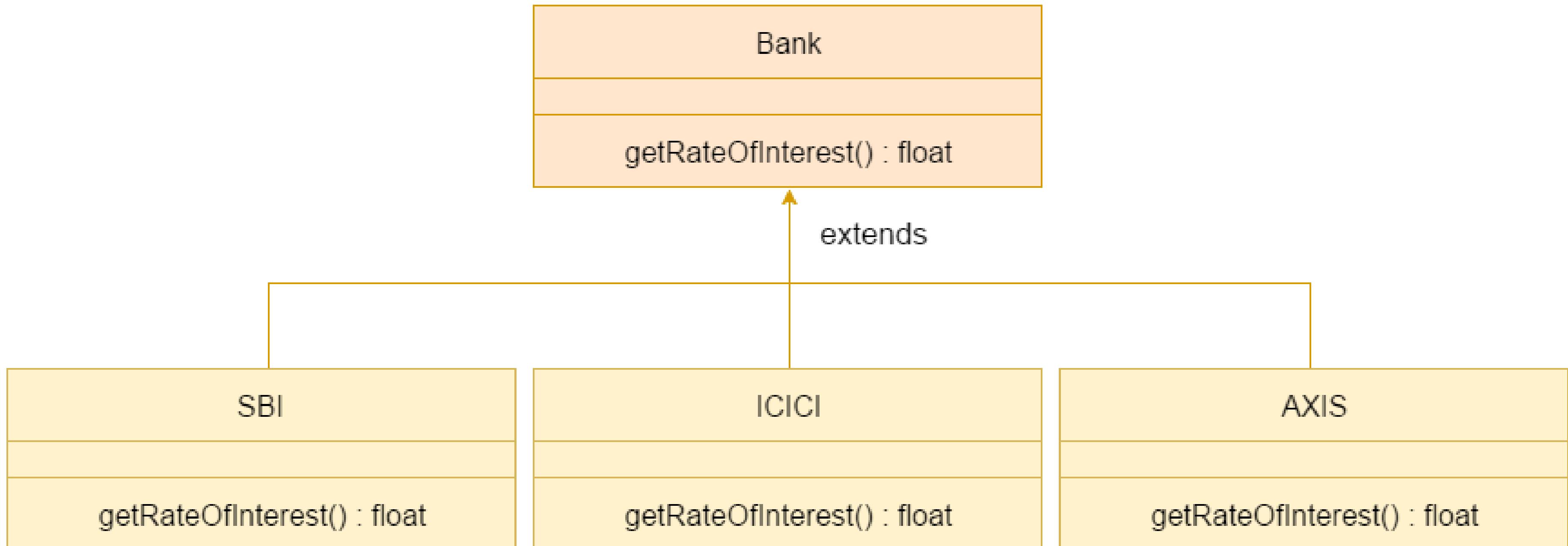
Quando un metodo in una sottoclasse ha lo stesso nome, gli stessi parametri o firma e lo stesso tipo restituito (o sottotipo) di un metodo nella sua superclasse, si dice che il metodo nella sottoclasse sovrascrive il metodo nella superclasse -classe.



```
1. classe Veicolo{  
2. // //Creazione di una classe genitore. definire un metodo  
3. void run(){System.out.println( "Il veicolo è in marcia" );}  
4.  
5.  
6. class Bike extends Veicolo{  
7. public static void main(String args[]){  
8. Bike obj = new Bike();  
9. obj.run();  
10. } }  
11.  
12. class Bike2 estende Veicolo{  
13. void run(){System.out.println( "La bici sta correndo in sicurezza" );} //metodo della classe genitore  
14. public static void main(String args[]){  
15. Bike2 obj = new Bike2(); //crea oggetto  
16. obj.run();  
17. } }
```

OVERRIDE del metodo in Java

Considera uno scenario in cui Bank è una classe che fornisce funzionalità per ottenere il tasso. Tuttavia, il tasso di interesse varia a seconda delle banche. esempio, le banche SBI, ICICI e AXIS potrebbero fornire tassi dell'8%, 7% e 9%.



```
1. class Bank{  
2.     int getRateOfInterest(){return 0;} }  
3. class SBI extends Bank{  
4.     int getRateOfInterest(){return 8;} }  
5. class ICICI extends Bank{  
6.     int getRateOfInterest(){return 7;} }  
7. class AXIS extends Bank{  
8.     int getRateOfInterest(){return 9;} }
```

```
1. class Test2{  
2.     public static void main(String args[]){  
3.         SBI s=new SBI();  
4.         ICICI i=new ICICI();  
5.         AXIS a=new AXIS();  
6.         System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());  
7.         System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());  
8.         System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());  
9.     }
```

```
1. class Bank{  
2.     int getRateOfInterest(){return 0;} }  
3. class SBI extends Bank{  
4.     int getRateOfInterest(){return 8;} }  
5. class ICICI extends Bank{  
6.     int getRateOfInterest(){return 7;} }  
7. class AXIS extends Bank{  
8.     int getRateOfInterest(){return 9;} }  
9.  
10. class Test2{  
11.     public static void main(String args[]){  
12.         SBI s=new SBI();  
13.         ICICI i=new ICICI();  
14.         AXIS a=new AXIS();  
15.         System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());  
16.         System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());  
17.         System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());  
18.     } }
```

```
1. class Bank{  
2.     int getRateOfInterest(){return 0;} }  
3. class SBI extends Bank{  
4.     int getRateOfInterest(){return 8;} }  
5. class ICICI extends Bank{  
6.     int getRateOfInterest(){return 7;} }  
7. class AXIS extends Bank{  
8.     int getRateOfInterest(){return 9;} }  
9.  
10. class Test2{  
11.     public static void main(String args[]){  
12.         SBI s=new SBI();  
13.         ICICI i=new ICICI();  
14.         AXIS a=new AXIS();  
15.         System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());  
16.         System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());  
17.         System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());  
18.     } }
```

I metodi **final** non possono essere sovrascritti : se non vogliamo che un metodo venga sovrascritto, lo dichiariamo **final**.

```
class Parent {  
    // Can't be overridden  
    final void show() {"mirko"} }  
class Child extends Parent {  
    // This would produce error  
    void show() {"non mirko"} }
```

I metodi statici non possono essere sovrascritti (**Method Overriding vs Method Hiding**): quando si definisce un metodo statico con la stessa firma di un metodo statico nella classe base, è noto come nascondere il metodo.

Metodo di istanza della
superclasse

Metodo statico della
superclasse

Metodo di istanza della
sottoclassile

Sostituzioni
Genera un errore in fase di
compilazione

Metodo statico della
sottoclassile

Genera un errore in fase di
compilazione

Nasconde

Il metodi privati non possono essere sovrascritti : i metodi privati non possono essere sovrascritti in quanto sono legati durante la compilazione. Pertanto non possiamo nemmeno sovrascrivere i metodi privati in una sottoclasse.

Il metodo di override deve avere lo stesso tipo di ritorno (o sottotipo): da Java 5.0 in poi è possibile avere un tipo di ritorno diverso per un metodo di override nella classe figlia, ma il tipo di ritorno del figlio deve essere un sottotipo del tipo di ritorno del genitore. Questo fenomeno è noto come tipo di ritorno covariante .

Override e costruttore: non possiamo sovrascrivere il costruttore poiché la classe genitore e figlia non può mai avere un costruttore con lo stesso nome (il nome del costruttore deve essere sempre lo stesso del nome della classe).

OVERRIDE

Invocare il metodo sovrascritto dalla sottoclasse: possiamo chiamare il metodo della classe genitore nel metodo di override usando la parola chiave super .

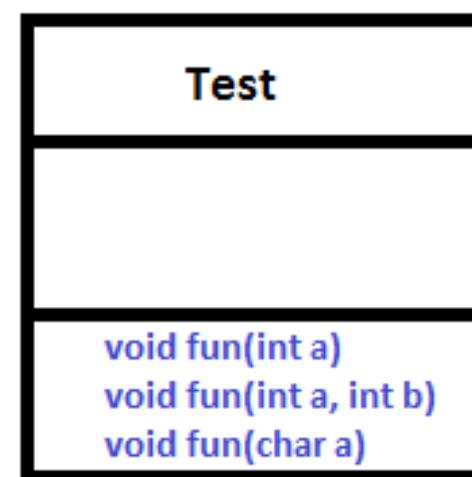
```
1.class Parent {  
2.    void show()  
3.    { System.out.println("Parent's show()"); }  
4.  
5.class Child extends Parent {  
6.    @Override  
7.    void show() {  
8.        super.show();  
9.        System.out.println("Child's show()"); }  
10.  
11.class Main {  
12.    public static void main(String[] args){  
13.        Parent obj = new Child();  
14.        obj.show();}  
15.}
```

OVERRIDE

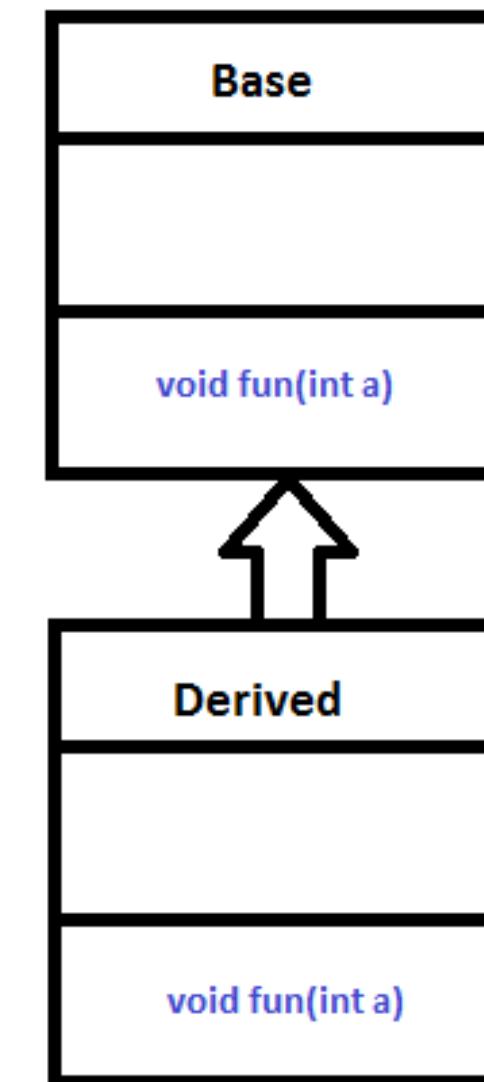
Override vs Overload :

Il sovraccarico riguarda lo stesso metodo con firme diverse. L'override riguarda lo stesso metodo, la stessa firma ma classi diverse collegate tramite ereditarietà.

L'overload è un esempio di polimorfismo in fase di compilazione e l'override è un esempio di polimorfismo in fase di esecuzione



Overloading



Overriding

Come affermato in precedenza, i metodi sovrascritti consentono a Java di supportare il polimorfismo in fase di esecuzione . Il polimorfismo è essenziale per la programmazione orientata agli oggetti per un motivo: consente a una classe generale di specificare metodi che saranno comuni a tutte le sue derivate, consentendo alle sottoclassi di definire l'implementazione specifica di alcuni o tutti quei metodi. I metodi sovrascritti sono un altro modo in cui Java implementa l'aspetto "un'interfaccia, più metodi" del polimorfismo.

Dynamic Method Dispatch è uno dei meccanismi più potenti che la progettazione orientata agli oggetti mette in campo per il riutilizzo e la robustezza del codice.

La capacità di esistere librerie di codice per chiamare metodi su istanze di nuove classi senza ricompilare mantenendo un'interfaccia astratta pulita è uno strumento profondamente potente. I metodi sovrascritti ci consentono di chiamare metodi di una qualsiasi delle classi derivate senza nemmeno conoscere il tipo di oggetto della classe derivata.

Esercizio Ereditarietà

Andiamo a creare una classe Utente per essere istanziata dovrà avere alcune caratteristiche quali (Nome:String Email:String Soldi:Float)

CONSIGLIO: e Una padre Ristorante con due array, piatti e valutazioni piatti

L'email e il Nome dovranno essere forniti dall'utente e il valore del credito(soldi) dovrà essere randomizzato, e rivalorizzato a ogni nuovo login

Creare un menu da cui sia possibile in loop creare un oggetto utente che ci farà inserire i dati necessari e alla fine del processo ci dia la possibilità di: far stampare i dati, ripetere l'operazione, interagire col profilo o uscire

Ora andiamo a creare due classi figlie, CHEF e CRITICO

Gli utenti potranno interagire col profilo, tramite il sistema che preferite (password, inserimento iniziale diverso, domanda x ecc) e diventare chef che potrà così aggiungere piatti, o CRITICO che potrà inserire le valutazioni.

AVANZATO: dopo 3 recensioni o inserimenti far diventare gli CHEF_CAPI e i CRITIC_FORTI dandogli la possibilità di stampare dai comandi una print unica

POLIMORFISMO

Il polimorfismo rappresenta il principio in funzione del quale diverse classi derivate possono implementare uno stesso comportamento definito nella classe base in modo differente.

Riprendendo l'esempio proposto nel caso dell'ereditarietà, consideriamo due comportamenti comuni a tutti gli animali: Respira e Mangia.

Nel mondo animale questi comportamenti vengono messi in atto secondo modalità peculiari a seconda delle specie: un carnivoro mangia in modo differente rispetto a un erbivoro, un pesce respira in modo diverso rispetto a un uccello. Sulla base di queste considerazioni, i comportamenti Mangia e Respira, definiti nelle diverse classi derivate da Animale, possono essere implementati in modo specifico e del tutto indipendente dalle altre implementazioni, in particolar modo, da quello della classe base Animale.

POLIMORFISMO

Tra i vari tipi di polimorfismo che possono essere presenti in un linguaggio, quello che si deve necessariamente ritrovare nella programmazione ad oggetti e' noto come polimorfismo di inclusione universale, basato proprio sul concetto di ereditarieta'.

Questa forma di polimorfismo si basa sui seguenti presupposti:

a) la possibilità di usare variabili polimorfe, cioe' che possono riferirsi ad oggetti di tipi diversi (generalmente "inclusi" in una certa gerarchia). In pratica basta permettere ad una variabile di tipo T di ricevere un oggetto di un qualsiasi sottotipo di T.

b) la possibilità di effettuare chiamate polimorfe, cioe' di indicare con lo stesso nome dei metodi che appartengono a classi diverse e che sono quindi generalmente diversi ("polimorfo" = "avente piu' forme").

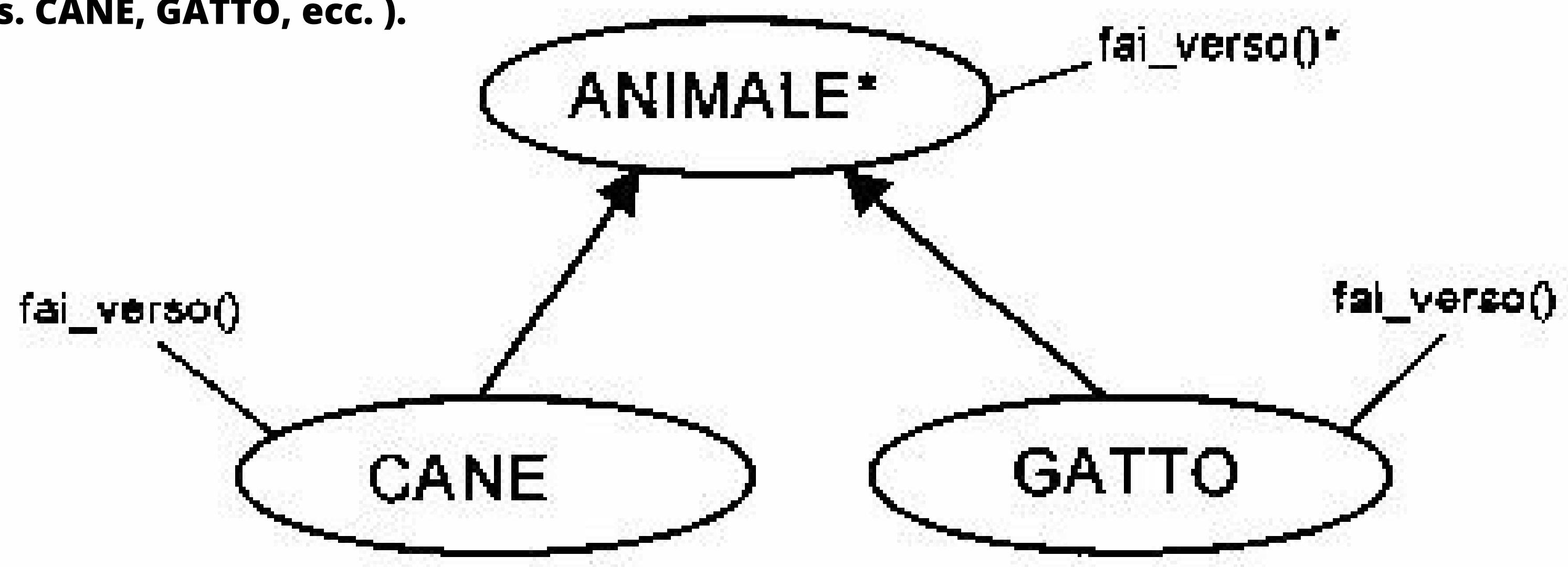
A differenza di altre forme di polimorfismo, come l'overloading degli operatori, (in italiano traducibile con sovraccarico o sovrapposizione) il polimorfismo di inclusione universale prevede che la decisione su quale debba essere la routine da richiamare viene presa a tempo di esecuzione a seconda della classe effettiva (piu' stretta) di appartenenza dell'oggetto rispetto a cui viene fatta la chiamata.

Questa tecnica e' nota come collegamento dinamico (late o dynamic binding) dei nomi al codice che deve essere effettivamente eseguito. Esso si contrappone al tradizionale collegamento statico (early o static binding) deciso dal compilatore, di norma, nel caso di chiamate non polimorfe.

POLIMORFISMO

Spesso il polimorfismo viene introdotto con le classi astratte, vale a dire classi in cui compaiono uno o piu' metodi la cui definizione viene rinviate alle sottoclassi (metodi astratti).

Tipico e' l'esempio rappresentato nella figura qui sotto, in cui la classe ANIMALE ha un metodo astratto fai_verso() che trova una concreta realizzazione (definizione) solo nei discendenti "concreti" (ad es. CANE, GATTO, ecc.).



POLIMORFISMO

In informatica, il termine polimorfismo viene usato in senso generico per riferirsi a espressioni che possono rappresentare valori di diversi tipi (dette espressioni polimorfiche).

In un linguaggio non tipizzato, tutte le espressioni sono polimorfiche

Il termine viene associato a due significati specifici:

- nel contesto della programmazione orientata agli oggetti, si riferisce al fatto che un'espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (polimorfismo per inclusione);**
- nel contesto della programmazione generica, si riferisce al fatto che il codice del programma può ricevere un tipo come parametro invece che conoscerlo a priori (polimorfismo parametrico).**

POLIMORFISMO

Polimorfismo significa "molte forme" e si verifica MASSIVAMENTE quando abbiamo molte classi correlate tra loro per ereditarietà o quando dobbiamo interagire con molte astrazioni.

Come abbiamo specificato nel capitolo precedente; L'ereditarietà ci consente di ereditare attributi e metodi da un'altra classe.

Il polimorfismo utilizza questi metodi per eseguire compiti diversi.

Questo ci permette di eseguire una singola azione in modi diversi.

Ad esempio, pensa a una superclasse chiamata Animal che ha un metodo chiamato animalSound().

Le sottoclassi di animal potrebbero essere maiali, gatti, cani, uccelli - E hanno anche la loro implementazione di un suono animale (il maiale grugnisce e il gatto miagola, ecc.):

ESEMPIO APPLICATO

```
1. class Animal {  
2.     public void animalSound() {  
3.         System.out.println("The animal makes a sound");  
4.     } }  
5. class Pig extends Animal {  
6.     public void animalSound() {  
7.         System.out.println("The pig says: wee wee");  
8.     } }  
9. class Dog extends Animal {  
10.    public void animalSound() {  
11.        System.out.println("The dog says: bow wow");  
12.    } }  
13. class Main {  
14.    public static void main(String[] args) {  
15.        Animal myAnimal = new Animal();  
16.        Animal myPig = new Pig();  
17.        Animal myDog = new Dog();  
18.        myAnimal.animalSound();  
19.        myPig.animalSound();  
20.        myDog.animalSound();  
21.    } }
```

POLIMORFISMO



Il polimorfismo si ha con un'azione combinata di compilatore e linker.

Al contrario di quanto accade nella maggior parte dei casi, il run-time ha un ruolo importantissimo nell'esecuzione di codice polimorfo, in quanto non è possibile sapere, a compile-time, la classe di appartenenza degli oggetti istanziati.

Il compilatore ha il ruolo di preparare l'occorrente per far decidere l'esecutore quale metodo invocare. Ai fini della programmazione polimorfa non è necessario conoscere il linguaggio assemblativo, è necessario come base sull'indirizzamento per capire .

Quando viene compilata la classe base, il compilatore identifica i metodi che sono stati dichiarati virtuali (parola chiave MustOverride in Visual Basic, virtual in C++ e simbolo "#" in progettazione UML), e costruisce una Tabella dei Metodi Virtuali, indicando le signature o firma delle funzioni da sottoporre a override. Queste funzioni restano quindi "orfane", non hanno cioè un indirizzo per l'entry-point. Quando il compilatore si occupa delle classi derivate, raggruppa i metodi sottoposti ad override in una nuova TMV, di struttura identica a quella della classe base, stavolta indicando gli indirizzi dell'entry-point.

Esercizio Polimorfismo

Andiamo a creare una classe MENU che può creare vari oggetti che sono definiti come piatti con tre ARGOMENTI l'uno (Ingredienti, Prezzo, Chef)

Andiamo a creare una classe ORDINAZIONE che può diventare qualsiasi tipo di piatto, ma con la sola caratteristica del prezzo del piatto per creare un calcolo unitario del totale che possa essere richiamato dalla classe ORDINAZIONE

ASTRAZIONE

La pratica consiste nel presentare il sistema, ad esempio un pezzo di codice sorgente o uno scambio di trasmissioni di dati, in maniera ridotta ai soli dettagli considerati essenziali all'interesse specifico, ad esempio raggruppando il codice in una funzione o formalizzando un protocollo di comunicazione.

Indica quanto il codice scritto in un linguaggio di programmazione si distacca dalle istruzioni in linguaggio macchina che ad esso corrisponderanno dopo l'operazione di compilazione. Delle istruzioni scritte in Java, per esempio, sono molto più vicine al linguaggio comprensibile all'uomo piuttosto che a quello comprensibile dalla macchina (alto livello di astrazione). Viceversa delle istruzioni scritte in Assembly sono abbastanza vicine alle istruzioni in formato comprensibile alla macchina (basso livello di astrazione).

La seguente definizione di astrazione aiuta a capire come questo termine viene applicato all'informatica:

«Astrazione - Un concetto o un'idea non associata a nessuna istanza specifica.»

ASTRAZIONE

L' astrazione dei dati è il processo che nasconde i dettagli e da all'utente solo le informazioni essenziali. L'astrazione può essere ottenuta sia con classi astratte che con interfacce

**La abstract come parola chiave è un modificatore di non accesso, utilizzato per classi e metodi:
Classe astratta: è una classe ristretta che non può essere utilizzata per creare oggetti (per accedervi deve essere ereditata da un'altra classe).**

Metodo astratto: può essere utilizzato solo in una classe astratta e non ha un corpo. Il corpo è fornito dalla sottoclasse (ereditato da).

Una classe astratta può avere sia metodi astratti che regolari:

```
abstract class Animal {  
    public abstract void animalSound();  
    public void sleep() {  
        System.out.println("Zzz");  
    } }
```

ESEMPIO APPLICATO

```
1. abstract class Animal {  
2.     public abstract void animalSound(); }  
3.  
4. class Pig extends Animal {  
5.     public void animalSound() {  
6.         System.out.println("The pig says: wee wee");  
7.     } }  
8. class Dog extends Animal {  
9.     public void animalSound() {  
10.        System.out.println("The dog says: bow wow");  
11.    } }  
12. class Main {  
13.     public static void main(String[] args) {  
14.         Animal myAnimal = new Animal();  
15.         Animal myPig = new Pig();  
16.         Animal myDog = new Dog();  
17.         myAnimal.animalSound();  
18.         myPig.animalSound();  
19.         myDog.animalSound();  
20.     } }
```

```
1. interface Animal {  
2.     public void animalSound();  
3.     public void sleep(); }  
4. class Pig implements Animal {  
5.     public void animalSound() {  
6.         System.out.println("The pig says: wee wee"); }  
7.     public void sleep() {  
8.         System.out.println("Zzz");}  
9.  
10. class Main {  
11.     public static void main(String[] args) {  
12.         Pig myPig = new Pig();  
13.         myPig.animalSound();  
14.         myPig.sleep();  
15.     } }
```