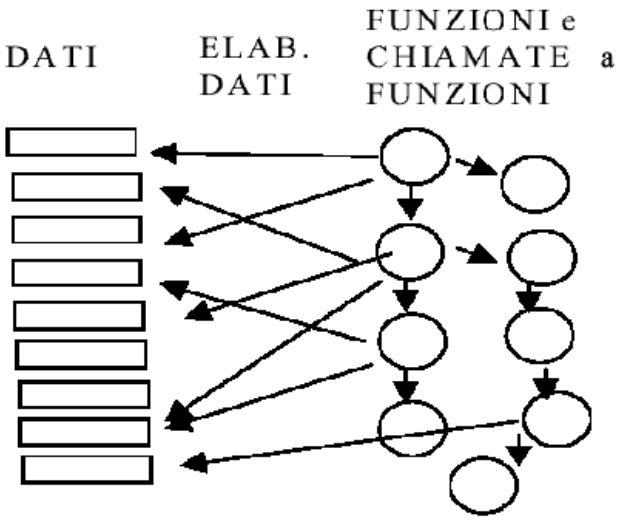
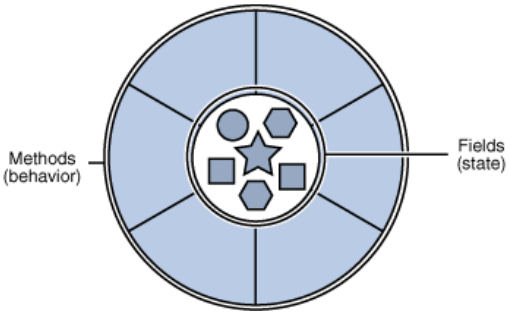
La programmazione: tradizionale vs Orientata agli Oggetti (OOP)

In generale, nella soluzione di un problema si individuano dati e funzionalità (azioni) potendola implementare, a livello più semplice, in un programma costituito da *istruzioni* che elaborano dati:

Tradizionale:



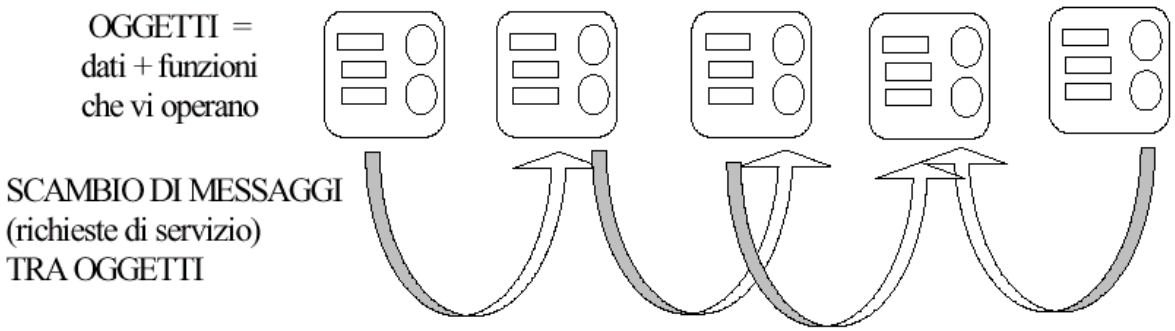
nella progettazione **OO** il codice che manipola i dati è invece ***incapsulato*** con la dichiarazione e memorizzazione di quei dati.



Si pensi all'incapsulamento come ad un involucro protettivo che avvolge sia le *istruzioni* che i dati che si stanno manipolando. Tale involucro definisce il comportamento e protegge da accessi arbitrari da parte di un altro programma; il pregio è che ognuno può accedere al codice e usarlo senza preoccuparsi dei dettagli di implementazione.

L'essenza della **programmazione Orientata agli Oggetti** è trattare come entità concrete *oggetti* anche astratti con un proprio specifico comportamento che *rispondono a messaggi* che dicono loro di fare qualcosa; una sequenza di passi di un procedimento tipica della programmazione con paradigma "procedurale" può diventare una collezione di messaggi tra oggetti autonomi.

Ad Oggetti:



Quindi, all'interno dell'impostazione *procedurale*, si introduce un ***GROSSO CAMBIAMENTO DI PARADIGMA:***

- **NON** si richiede a funzioni di operare su dati
  nome_funzione (parametro1, parametro2, ….);
- **MA** si richiede alle entità di eseguire delle funzioni
  oggetto.nome_funzione (parametro1, parametro2, ….);

MODELLO CLIENTE/SERVITORE, quindi, con spostamento del "fuoco"

- **NON** chiamare una funzione che elabori dei dati
- **MA** chiedere a una entità software di svolgere un servizio

Esempio: dato un flusso, NON invocare una funzione che scriva dati sul flusso,
MA chiedere al ***flusso*** stesso di svolgere il servizio relativo alla **scrittura**

***System.out*.println(*stringa*)**

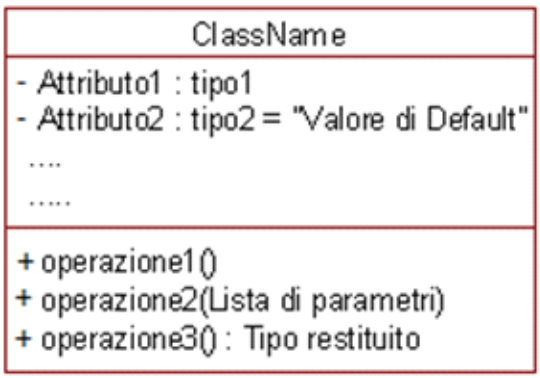
- I clienti *non conoscono* l'organizzazione interna dei centri di servizio, e *non possono* accedere *direttamente* a essa
- non interessa l'algoritmo (come sono fatte le cose) ma interessa che le *cose vengano fatte*
- i clienti sono indipendenti da come sono fatti i servitori: facilita la manutenzione!

Nascono metodi di progettazione OO ad esempio quello proposto da Abbot, Booch e Lorensen: tecnica che suggerisce di **evidenziare nel testo** gli elementi candidati a diventare, nel progetto, una **classe** (testo evidenziato con bordo), **attributi** o **proprietà** (testo evidenziato con sottolineatura punteggiata) e il **comportamento** (testo evidenziato con sottolineatura) delle entità software (oggetti – esemplari di quella classe/categoria). Il comportamento è realizzato con **metodi**, ovvero sottoprogrammi che elaborano i dati memorizzati nelle proprietà per produrre nuove informazioni.

Nasce un linguaggio di modellazione: l'**UML** che permette di descrivere *astrattamente* una **classe** con un nome **ClassName** e con **proprietà** e **metodi** che saranno comuni a tutti gli esemplari (oggetti o *istanze cioè realizzazioni concrete con occupazione di memoria RAM)*:

- **proprietà private** (*incapsulate nella classe*)
- **metodi pubblici**:
  - che permettono di accedere alle proprietà private (*metodi di accesso*)
  - che definiscono l'elaborazione (operazioni proprie delle varie *istanze*)

Nell'illustrare il **diagramma statico di una classe**::



Il simbolo – specifica che l'accesso è privato
(***specificatore di accesso privato***)

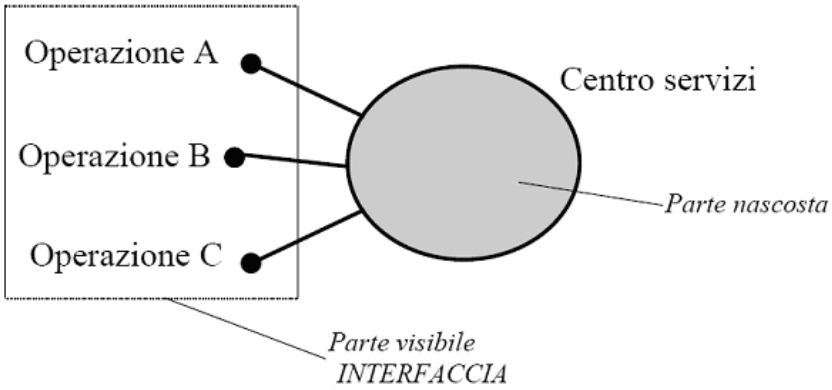
Il simbolo + specifica che l'accesso è pubblico
(***specificatore di accesso pubblico***)

Nell'esempio si progettano gli attributi "***nascosti***": *visibili solo dai metodi della classe*
mentre i metodi si pensano pubblici: *tutti possono vederli/usarli*

*Visibilità: vedremo, in seguito, diversi **modificatori di accesso***

IL CONCETTO DI OGGETTO

- Un oggetto è inteso come un ***centro di servizi***, un'***astrazione di dato***
  - con una *parte visibile* → ***INTERFACCIA***
  - ed una *parte nascosta*



LE OPERAZIONI DI *INTERFACCIA* RAPPRESENTANO:

- I servizi che gli altri oggetti possono richiedere all'oggetto
- Modalità di trattamento dei dati dell'oggetto o di accesso ai dati

LA *PARTE NASCOSTA* CONTIENE:

DATI che sono accessibili solo attraverso richieste di servizio:

- Gli attributi che caratterizzano l'oggetto
- Lo stato dell'oggetto

OPERAZIONI PRIVATE, sfruttate dalle operazioni pubbliche ma non servizi esterni

DESCRIZIONE DI UN SISTEMA A OGGETTI

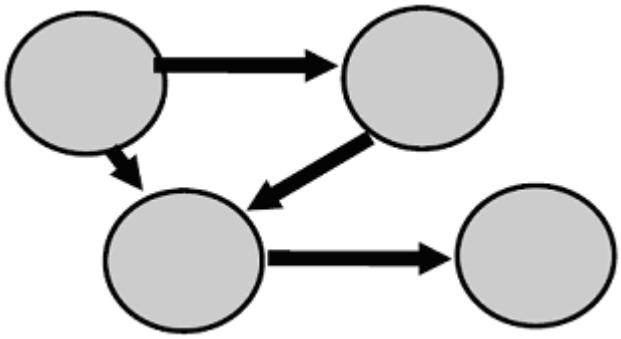
Ogni oggetto appartiene a ("è istanza di") una data ***classe***
La classe *racchiude e incapsula la specifica* di:

- **struttura** dell'oggetto (dati)
- **comportamento** dell'oggetto (operazioni)

Le classi possono essere correlate tra loro (*tassonomie* di ***ereditarietà***) in modo da permettere il collegamento tra classi simili per raggruppare le proprietà simili e, in ultima analisi, per ***riusare facilmente*** il codice comune alle classi correlate e ridefinire velocemente nuove classi

ARCHITETTURA DI UN SISTEMA A OGGETTI

- Un ***insieme di oggetti***
    - che **interagiscono gli uni con gli altri**
    - senza conoscere nulla delle rispettive rappresentazioni concrete



VANTAGGI[1] DELLA PROGRAMMAZIONE AD OGGETTI

Facilitazione di costruzione *cooperativa* di software:

- diverse persone sviluppano diverse classi
- ogni programmatore può semplicemente verificare il comportamento delle sue classi
- istanziandone oggetti e verificandone il comportamento in risposta a richieste di servizio
- unico accordo necessario per integrare il tutto in un unico sistema finale: definire le interfacce delle classi

Facilitazione della *gestione e manutenzione*

- Se vi sono errori sui dati in un oggetto, è facile scoprire dove si trova l'errore, perché (siccome i dati non sono visibili all'esterno dell'oggetto) esso non potrà essere che all'interno dell'oggetto che gestisce quei dati
- le modifiche a una classe non rendono necessario modificare il resto del programma (le altre classi) a meno che non venga modificata l'interfaccia

Supporto a progettazione e sviluppo *incrementali* : si possono definire nuove classi sfruttando il codice di classi già esistenti

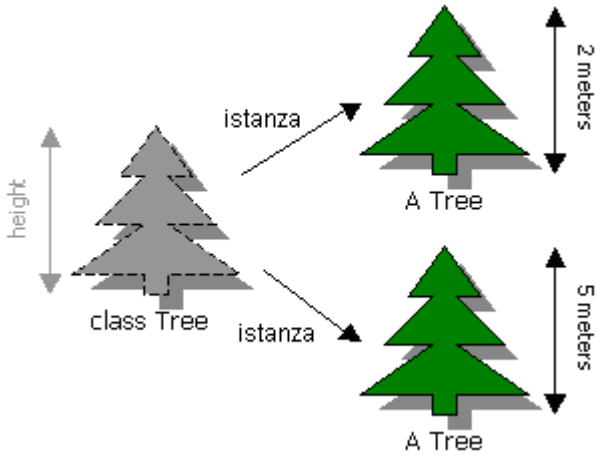
Possibilità di *rapida prototipazione* : non importa che una classe sia completamente definita per poter iniziare a "testare" il funzionamento degli oggetti di quella classe.

[1] Interessante lettura "Linguaggi e Metodologie Orientate agli Oggetti" (*Object Oriented Analysis* ed *Object Oriented Design*)

Principi su cui si basa un linguaggio ad oggetti

Ecco quindi quelli che generalmente sono considerati i principi su cui si basa un **linguaggio ad oggetti**:

1. "Ogni cosa è un oggetto". Per comprendere meglio tale affermazione si può pensare all'oggetto come ad una *particolare variabile*. Particolare perchè riceve *dati* ma può anche compiere *operazioni* su sé stessa.



Un generico albero può essere caratterizzato dall'**altezza** (`height` variabile nel tempo)

e dal ***metodo*** che determina la sua crescita ad esempio di un metro

2. "Un programma è una serie di oggetti in comunicazione tra di loro". La comunicazione tra gli oggetti avviene principalmente attraverso *messaggi*. Scendendo più nel dettaglio, si può pensare ad un messaggio come ad una *richiesta* fatta all'oggetto di richiamare una funzione di competenza dello stesso.

3. "Ciò che risulta dall'insieme di più oggetti è esso stesso un oggetto". La questione riflette in maniera abbastanza esplicita il riferimento al punto 1: se "ogni cosa è un oggetto" anche tanti oggetti messi insieme sono un oggetto.

4. "Ogni oggetto appartiene a un tipo definito" ossia ogni oggetto è un'istanza di una classe.

> Per adesso basterà sapere che una ***classe*** è il tipo (o la categoria) di oggetto, ad esempio la categoria delle auto, delle moto, delle navi e così via.
>
> L'***istanza*** è invece un particolare elemento della categoria. Prendiamo l'esempio delle automobili: la classe rappresenta il concetto astratto di "automobile". L'istanza è per esempio la propria macchina,
>
>
>
> quella del vicino di casa, e così via.

5. "Tutti gli oggetti facenti parte della stessa categoria possono ricevere i medesimi messaggi" questo significa che entrando in due macchine diverse si può essere sicuri di avere la possibilità di accendere i fari perchè entrambe le automobili supportano questo tipo di **messaggio 'accendi i fari'**. In generale quindi se si ha a che fare con un elemento dell'insieme 'automobili' si può compiere ogni operazione consentita dalla categoria.

6. "Ogni oggetto è dotato di un'interfaccia" oggetti del tutto identici tra loro tranne che per lo stato che possono assumere (o che assumono) nell'esecuzione di un'applicazione (si pensi ad esempio a due lampadine, una accesa ed una spenta) sono raggruppabili come già detto in classi di oggetti.

Una volta che una classe è definita, è possibile creare tanti oggetti (ossia tanti elementi della classe) quanti ne servono per lo sviluppo dell'applicazione.

Ma come si può manipolare uno di questi oggetti? Ogni volta che chiediamo ad un oggetto di compiere una determinata operazione, interagiamo con l'interfaccia dell'oggetto, differente in base alla classe alla quale esso appartiene. Si pensi ad esempio ad una lampadina, come nella figura sottostante:

La categoria delle lampadine, ossia la classe, è chiamata 'Lampadine'. La particolare lampadina che prendiamo in considerazione è quella del salotto di casa, e ciò che può essere fatto con essa (in parole povere il suo *scopo*) è *definito dall'interfaccia*. La lampadina può allora essere accesa, spenta o diffondere una luce soffusa.

Ad esempio, riprendendo l'esempio dell'albero, in linguaggio Java si creerà una classe di nome `Tree` con un attributo (`height`) nascosto, inizialmente nello *stato* `height=0` e come interfaccia, il metodo che modifica tale dato incrementandolo di uno:

```
class Tree {

                /**
                    altezza dell'albero
                */
                private int height;     // attributo nascosto
                                        // inizializzato di default a zero
                /**
                    Crescita di un metro
                */
                public void cresci() {

                          height = height + 1;
                }

                public static void main(String [] args) {

                        // Creazione di un oggetto della classe Tree

                        Tree tree1 = new Tree();

                        // richiesta di crescere di un metro

                        tree1.cresci();
                }
}
```

OOP: evoluzione del paradigma procedurale implementato in Java

==========

L'essenza, dunque, della **programmazione Orientata agli Oggetti** è trattare come entità concrete oggetti anche astratti con un proprio specifico comportamento che rispondono a messaggi che dicono loro di fare qualcosa; una sequenza di passi di un procedimento può diventare una collezione di messaggi tra oggetti autonomi.

Il paradigma OO traduce il ***pensare per categorie*** e tende a creare delle "unità" indipendenti di codice (gli *oggetti*) dotate in un certo senso di vita propria, come gli oggetti del mondo reale, che possano essere riutilizzate il più possibile all'interno dello stesso programma o in programmi diversi.

Gli *oggetti* vengono creati partendo da modelli più generali denominati *classi*.

Una ***classe*** è in pratica una famiglia di oggetti, di cui contiene tutte le caratteristiche e i comportamenti comuni.

Le classi sono la teoria, gli oggetti ne costituiscono le ***istanze***, ovvero la realizzazione concreta.

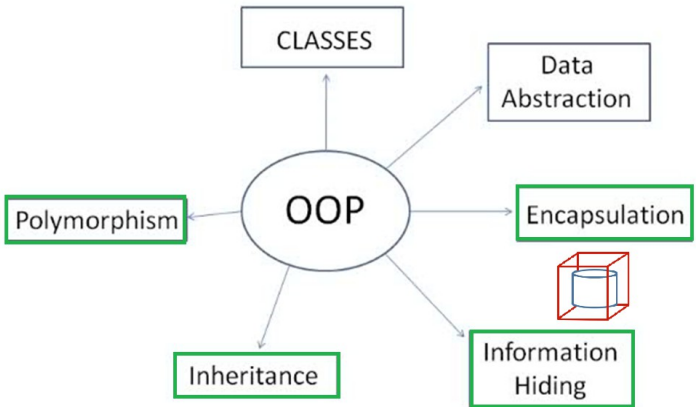
Una classe descrive le "caratteristiche generali" di oggetti e può essere vista come il mattone per implementare algoritmi (in Java un programma è una classe): una scatola che ingloba dati (***attributi*** degli oggetti) e funzioni (***metodi***) e classifica con un nome una collezione di oggetti.

classe	*descrizione astratta di una categoria di oggetti che ne illustra le caratteristiche ed i comportamenti comuni*
oggetto	*istanza di una classe cioè realizzazione concreta con occupazione di memoria (RAM)*

Siamo abituati a classificare col nome (astratto) ad esempio "Cane" un animale con 4 zampe (*attributo*) che abbaia (*metodo*); a tale classe apparterrà il cane lupo Rintintin (istanza di un oggetto che occuperà memoria).

==========

Riassumendo, i meccanismi che rinforzano il modello object-oriented sono: l'***incapsulamento***, l'***ereditarietà*** e il ***polimorfismo***.



INCAPSULAMENTO

A livello più semplice ogni programma è costituito da istruzioni e dati: nella progettazione OO ***il codice che manipola i dati è incapsulato con la dichiarazione e memorizzazione di quei dati***.

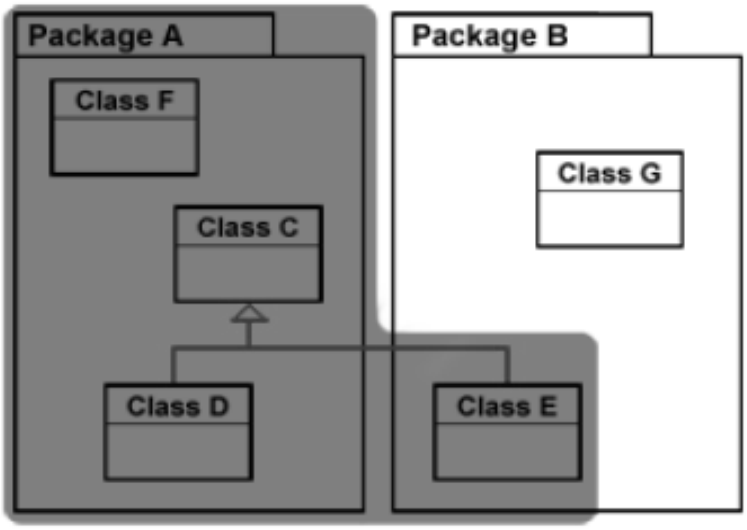
Si pensi all'incapsulamento come ad un involucro protettivo che avvolge sia le istruzioni che i dati che si stanno manipolando. Tale involucro definisce il comportamento e protegge da accessi arbitrari da parte di un altro programma; il pregio è che ognuno può accedere al codice e usarlo senza preoccuparsi dei dettagli di implementazione.

Base dell'incapsulamento è la *classe*.

Una classe è costituita da ***attributi*** registrati in variabili il cui valore può variare da un oggetto all'altro e che per questo sono definite *variabili istanza.* Gli oggetti di una classe dispongono di *comportamenti* (immutati per tutti gli oggetti della classe), detti *metodi,* che nella programmazione corrispondono in pratica alle funzioni.

Una classe ha, dunque, sia dati che funzioni associate: i dati sono chiamati "*dati membro*" e le "*funzioni membro*" sono anche dette ***metodi***. L'ambito o **visibilità** di tali membri può essere controllato coi seguenti *modificatori* in fase di dichiarazione:

- **public** sono membri accessibili in tutto lo spazio di visibilità dell'oggetto in quella classe (visibili sia all'interno che all'esterno della classe in cui sono definiti ed a tutti gli altri metodi: è il *livello di protezione* **più ampio** da usarsi con moderazione ma in certi casi è obbligatorio ad esempio per dichiarare la classe di un applet)
- **private** sono membri accessibili solo attraverso funzioni membro (è il *livello di protezione* **più ristretto**: visibili solo alla classe in cui sono dichiarati. Per buona programmazione si consiglia che i membri siano private per non modificarli accidentalmente o meno in modo illecito)
- **protected** sono membri che si comportano come privati ai fini della classe base ma come pubblici per le classi *derivate* e solo per queste (visibili all'interno del *package* in cui sono dichiarati e alle *sottoclassi* della classe origine). Nella figura sottostante, all'interno dell'area con sfondo grigio, un'esemplificazione di classi derivate definite anche in package diversi.
- livello di **default** viene abilitato se non si usa nessun modificatore: ambito costituito dal ***package*** di cui fa parte la relativa classe (visibilità alle classi dello stesso *package*)



Una **classe** è una descrizione astratta di un insieme di **oggetti** e pertanto non riserva area di memoria; le istanze di altri tipi (e non di classe) sono le **variabili**.

Un programma in Java è una classe e può essere usata in due modi:

- Aggiungendo un metodo *main* per poter eseguire direttamente l'***applicazione***
- Creando un ***applet*** che usi la classe e sarà attivato da un browser

Per creare una classe in Java, basta specificarne il nome in una dichiarazione ***class***:

es: class Cane {}

Per creare in Java un oggetto (esemplare) o realizzare un' ***istanza*** di una classe occorre un'adeguata dichiarazione:

```
es: Cane mio = new Cane() ;    // crea un oggetto di tipo "cane" il cui identificatore è "mio".
                               // con richiamo al metodo costruttore
```

Analogamente, in forma non compatta:

```
Cane mio;                // crea un riferimento ad un oggetto del tipo della classe Cane
mio = new Cane();        // uso di operatore new per allocare memoria
                                    con sintassi   nomeOggetto = new NomeClasse(parametri)
```

*Il **costruttore** è un metodo particolare usato per inizializzare un nuovo oggetto. Se lo sviluppatore non dichiara un costruttore per una classe, ne viene creato uno automaticamente da Java: esso coincide con il costruttore della super-classe.*

Possono esistere più costruttori: metodi pubblici, senza tipo, col nome della classe.

*Il **costruttore di default non prevede passaggio di parametri***; gli altri sono detti ***parametrici*** appunto perché prevedono il passaggio di parametri. Solitamente un costruttore, allo scopo di istanziare un oggetto, imposta il valore dei campi-dati della classe in modo corretto.

Per approfondimenti consulta i lucidi realizzati dal Prof. Franco Zambonelli (Università degli Studi di Modena e Reggio Emilia Facoltà di Ingegneria – CORSO DI RETI DI CALCOLATORI) in collaborazione con Ing. Enrico Denti dell'Università di Bologna (Anno Accademico 2001-2002) con anche ulteriori approfondimenti sul concetto di classe.

Per **caratterizzare una classe** si definiranno:

- le *variabili istanza*: dichiarate al di fuori dei metodi, solitamente subito dopo la prima riga della dichiarazione di classe. Ad es:

```
class Cane  {
                String nome;  // di defaut public
 }
```

- le *variabili di classe*: visibili a tutte le istanze della classe (registrano le caratteristiche generiche, comuni a tutti gli oggetti della classe). Ad es:

```
class Cane  {
                static int numZampe;     // uso static
}
```

- le *costanti*: nomi assegnati a valori fissi che non variano durante l'esecuzione del programma (per migliorare la leggibilità del codice). Ad es:

```
class Cane  {
    final String messaggio = "Attenti!";   // uso final
}
```

- i ***metodi***: per definire i comportamenti degli oggetti di una classe (le operazioni che possono svolgere). La loro struttura è la seguente:

accesso modificatori tipoRisultato nomeMetodo (parametri) { corpo del metodo }

es: public static void main (String args []) {…..}

I metodi di classe (disponibili per tutte le istanze di una classe), come per le variabili di classe, si definiscono con la parola chiave ***static*** (non possono usare this).

Un esempio di metodo che ritorna un valore intero, risultato di un'elaborazione (ad esempio acquisisce il numero di anni), può essere il seguente (di default l'accesso è *public*):

es: int acquisisciAnni () {…}

I parametri di tipi primitivi vengono passati *per valore*, oggetti ed array invece *per riferimento*

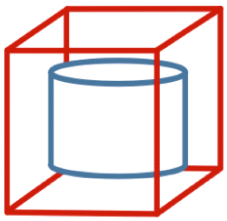
La chiamata ad un metodo (che sia *public*) può avvenire in Java con l'operatore punto '.'

es: mio.abbaia(); // nome dell'oggetto a sinistra del punto

Con attenzione all'**information hiding**, i metodi più significativi sono quelli (*public*) che permettono l'accesso, in modo controllato, ai dati nascosti (*private*) detti appunto ***metodi di accesso***: solitamente definiti per ogni attributo della classe, assumono la forma seguente

tipoAttributo getAttributo () {return *nomeAttributo*;}

void *setAttributo*(*tipoAttributo param*) { *nomeAttributo* = *param*;}



Ad esempio, se si è definito private il numero di anni:

```
class Cane  {
        private int anni;
        public int getAnni(){return anni;}
        void setAnni(int num) { // controllo sul parametro passato
                                    anni = num;}
        }
```

Nell'applicazione che usa tale classe si potrà accedere, modificandolo, all'attributo nascosto:

```
....
public static void main(String [] args) {

        // Creazione di un oggetto della classe Cane

        Cane mio = new Cane();

        // richiesta di crescere di un anno

        mio.setAnni(1);
}
```

In Java si utilizza la notazione puntata (*dot notation*) anche con le parole chiave *this* (oggetto corrente) e *super* (superclasse) usate per evitare problemi di interpretazione se esistono variabili con stesso nome.

Ad esempio: **this**.nomeVar // fa riferimento a nomeVar dell'oggetto corrente
super.nomeVar // fa riferimento a variabile con lo stesso nome della superclasse

EREDITARIETÀ

Ogni classe derivata eredita tutte le proprietà della classe base di partenza.

Le classi che ereditano metodi o attributi da un'altra vengono dette *sottoclassi* di quest'ultima che è chiamata *superclasse*. In Java l'ereditarietà è automatica: una sottoclasse eredita tutte le variabili e metodi della sua superclasse: tutto ciò che serve è definire le caratteristiche che la distinguono dalle precedenti, procedendo per *specializzazione*.

Per definire una sottoclasse in Java si usa la parola chiave ***extends*** :

es: class Spinone extends Cane {}

In Java esiste una classe predefinita che rappresenta il vertice della gerarchia ereditaria di classi:

Object

Quando si crea una nuova classe senza specificare la parola chiave extends, questa viene considerata una sottoclasse di Object, che rappresenta perciò la "madre di tutte le classi".

Java è un linguaggio ad *ereditarietà singola* nel senso che una classe può avere un'unica superclasse (gli sviluppatori hanno scelto di rendere più semplice il linguaggio).

Si realizza con le ***interfacce*** la possibilità di assegnare a classi diverse comportamenti simili: per ogni classe è possibile implementare un numero qualsiasi di interfacce (contenitori che contengono i nomi di metodi in essa contenuti ma non le relative definizioni) e fare in modo che classi distinte implementino la stessa interfaccia e rispondano agli stessi metodi (anche se in modo diverso).

Per definire una interfaccia in Java si usa la parola chiave ***interface*** :

es: public interface Interfaccia {}

ed è possibile usare la parola chiave extends per specificare che un'interfaccia eredita da un'altra.

es: public interface Interfaccia extends InterfacciaMadre {}

Se una nuova classe implementa un'interfaccia si usa la parola chiave ***implements****:*

es: classNuovaClasse implements Interfaccia {}

In molti casi è possibile usare le interfacce al posto delle classi (anch'esse compilate in file con estensione *.class*) purchè i relativi metodi siano definiti altrove.

Esempio di una classe che implementa più interfacce :

es: public class UnaClasse implements Interfaccia1, Interfaccia2, Interfaccia3;

ed in seguito si definiscono i metodi.

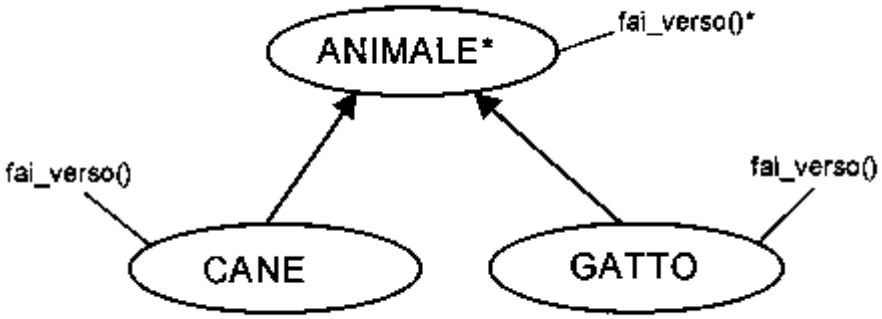
Per approfondimenti consulta i lucidi realizzati dal Prof. Franco Zambonelli (Università degli Studi di Modena e Reggio Emilia Facoltà di Ingegneria – CORSO DI RETI DI CALCOLATORI) in collaborazione con Ing. Enrico Denti dell'Università di Bologna (Anno Accademico 2001-2002)

POLIMORFISMO

Diverse classi possono svolgere le stesse azioni in modo differente.

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***

Ad esempio il metodo **fai_verso()**
per un cane sarà abbaiare,
per un gatto miagolare.



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*** cioè ***ridefinizione*** di metodi in sottoclasse oppure, anche nella stessa classe, operazioni di ***overloading*** cioè definizione di metodi con lo stesso nome ma differente *segnatura* (detta anche *firma*) cioè lista di paramentri (diverso numero e/o tipo di parametri); sono tipico esempio di overloading i diversi costruttori di una classe. Il tipo di ritorno non fa parte della f*irma* di un metodo, quindi non ha importanza per l'argomento overload. (In alcuni testi, l'overload non è considerato come aspetto polimorfico di un linguaggio.)

È bene notare che ci sono delle regole da rispettare per l'**override**:

1) se si decide di riscrivere un metodo in una sottoclasse, dobbiamo utilizzare la **stessa identica segnatura**, altrimenti utilizzeremo un overload in luogo di un override.

2) il **tipo di ritorno** del metodo **deve coincidere** con quello del metodo che si sta riscrivendo.

3) il metodo ridefinito, **non deve essere meno accessibile** del metodo che ridefinisce. Per esempio se un metodo ereditato è dichiarato protetto, non si può ridefinire privato, ma semmai, pubblico.

Per visualizzare le slides presentate nel corso.

Per esempi anche in altri linguaggi.

In Java esistono anche operatori *polimorfi* (dal greco "molte forme"): ad esempio l'operatore con il simbolo +

A seconda del contesto, tale operatore agisce diversamente:

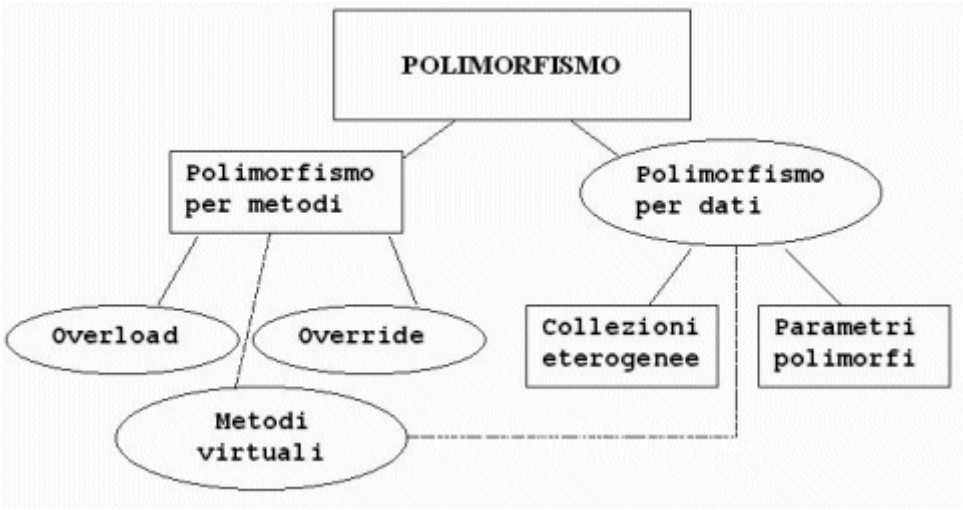
- l'operatore + **somma** operandi di tipo numerico:

```
int ris = num1 + num2;  // somma
```

- l'operatore + **appende** ad una stringa anche caratteri e numeri, si pensi agli output dell'esempio sottoriportato:

```
System.out.println("Il numero " + num + " è divisore" ); // concatena
```

Utilizzando una convenzione con la quale rappresenteremo con rettangoli i concetti, e con ovali concetti che hanno una reale implementazione in Java, cercheremo di schematizzare il polimorfismo e le sue espressioni.



==========

Package

Un *package* di Java è una sorta di "pacchetto" in cui sono riunite classi e interfacce (al fine della riusabilità). Usando i package si possono fornire gruppi di classi utilizzabili per svolgere un determinato compito, evitando che si verifichino conflitti tra nomi di classi, metodi o variabili di gruppi diversi (si creano sostanzialmente delle *librerie*).

Per definire una package in Java si usa la parola chiave ***package*** :

es: package Esempio;

e tale istruzione deve trovarsi sulla prima riga effettiva di un file di codice sorgente (eventualmente dopo righe vuote e commenti).

Per usare le classi contenute in un package si può procedere in due modi:

- importare il package all'inizio del file sorgente e riferirsi direttamente ai nomi delle classi desiderate:
  es: import java.io.*; // per importare tutte le classi del package
  import nomePackage.NomeClasse; // per importare solo NomeClasse

- indicare il nome completo del package prima del nome di classe, separandolo con un punto

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

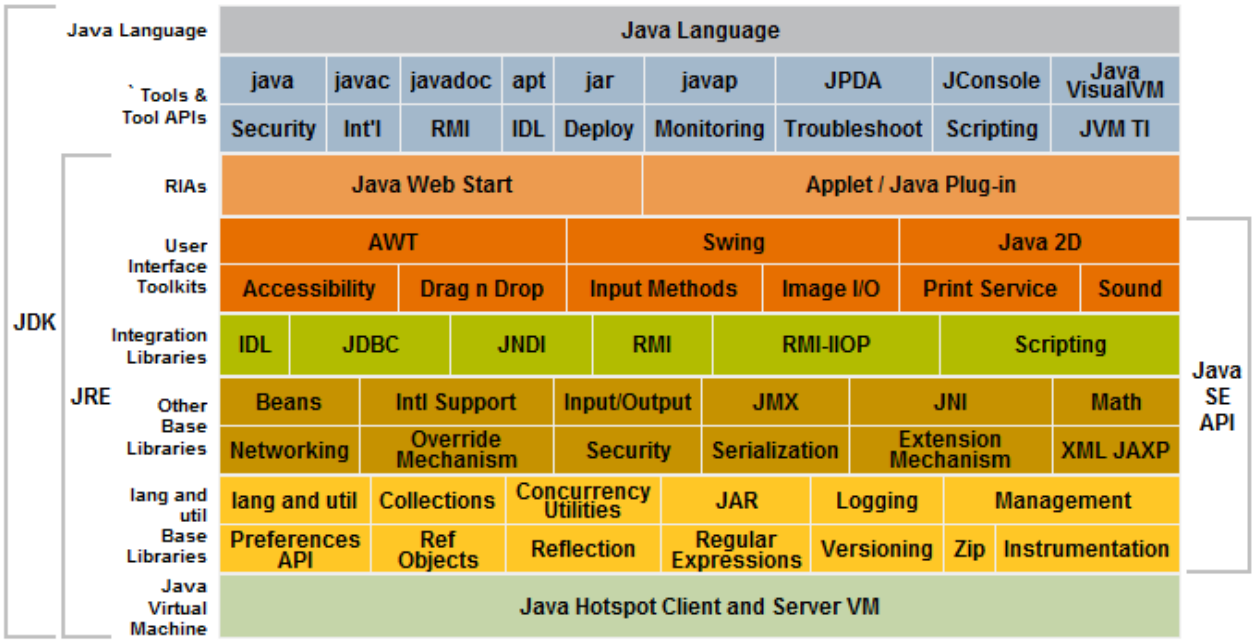
Java Platform, Standard Edition 8 API Specification

Il JDK (kit di sviluppo Java) è fornito con un'ampia libreria di classi predefinite, raccolte in una serie di package tutti contenuti nel package *java.* Tra i package disponibili:

- java.lang (classi relative al linguaggio, importato automaticamente)
- java.util (classi di utilità generale)
- java.io[2] (classi per operazioni di input e output)
- java.net (classi per operazioni di rete)
- java.awt (classi per implementare l'interfaccia utente: *Abstract Windowing Toolkit*)
- java.applet (classi per implementare applet)

---

documentation



[2] Per approfondimenti sul package di IO consulta i lucidi realizzati dal Prof. Franco Zambonelli (Università degli Studi di Modena e Reggio Emilia Facoltà di Ingegneria – CORSO DI RETI DI CALCOLATORI) in collaborazione con Ing. Enrico Denti dell'Università di Bologna (Anno Accademico 2001-2002). Per introdurre il concetto di eccezione.