Java™

Dott. Antonio Giovanni Lezzi

JAVA

- Fondamenti
- Programmazione Object Oriented
- Java e OOP nel dettaglio
- Java Avanzato
- Network
- Database
- Web

Dott. Antonio Giovanni Lezzi

Afol Milano

PROGRAMMAZIONE OBJECT ORIENTED

- Classi: definizione attributi e metodi di classe con costruttore
- Introduzione alla programmazione ad oggetti e progettazione (OO)
- Ereditarietà e sue applicazioni
- Classi e metodi abstract
- Interfacce e loro impiego
- Polimorfismo e sue applicazioni pratiche

Dott. Antonio Giovanni Lezzi Afol Milano

CLASSE ED OGGETTI

- **Classe** è una collezione di uno o più oggetti contenenti un insieme uniforme di attributi e servizi, insieme ad una descrizione circa come creare nuovi elementi della classe stessa;

- Un **Oggetto** è dotato di stato, comportamento ed identità; la struttura ed il comportamento di oggetti simili sono definiti nelle loro classi comuni; i termini istanza ed oggetto sono intercambiabili.

MODELLARE LA REALTÀ

STATO
via1: verde
via2: rosso

STATO
motore:acceso
velocità: 0

COMPORTAMENTO
Parti!
Frena!
Sterza!

METODI

- Tutti i linguaggi di programmazione forniscono la possibilità di definire, sotto un nome, gruppi di istruzioni / insiemi di linee di codice / blocchi di espressioni
- Possiamo riutilizzare un blocco di codice in molte parti del programma, semplicemente richiamando il nome con cui l'abbiamo definito, senza dover riscrivere tutto il blocco ogni volta.
- Questi aggregati, a seconda del linguaggio, si chiamano funzioni, procedure, subroutines o sottoprogrammi.
- In Java utilizziamo la logica definita dai blocchi di istruzioni per rappresentare il comportamento di classi di oggetti, prendono il nome di **metodi**.

METODI IN JAVA

La sintassi per la definizione di un metodo è simile alla definizione di una variabile:

```
[public|protected|private] [static] [final] Tipo
    identificatore([Tipo1 parametro1, Tipo2 parametro2, ...,
TipoN parametroN])
    [throws Eccezione1, Eccezzione2, ...] {

        // blocco di codice appartenente al metodo

    return varTipo;
}
```

METODO: I PARAMETRI

La sezione ***[Tipo1 parametro1, ... TipoN parametroN]*** è detto insieme dei parametri (formali) del metodo in cui si dichiara il tipo ed il nome simbolico delle variabili che il blocco di codice dovrà ricevere dal programma chiamante per poter svolgere il proprio compito.

```
public double areaRettangolo(double base,double
altezza){
    double a = base * altezza;
    return a;
}
```

METODO: VALORE DI RITORNO

- Il valore specificato accanto a return deve essere del medesimo tipo specificato nella dichiarazione del metodo ma deve essere omesso se il metodo è stato dichiarato come void.

- Traducibile come "vuoto" dichiarare un metodo void significa dire che il metodo non ritornerà alcun valore ed in tal caso la keyword return può essere anche omessa.

```
public double areaRettangolo(double base, double altezza) {
     if(base == 0.0 || altezza == 0.0)
        return 0.0; // questo return causa l'uscita dal metodo

    double a = base * altezza;

    return a;
}
```

METODI: IL RICHIAMO

- Per chiamare il nostro metodo all'interno della classe in cui lo stiamo definendo (fuori dalla classe di definizione occorre una sintassi differente, che vedremo in seguito), sarà sufficiente scrivere qualcosa del genere:

```
double areaCalcolata;

areaCalcolata = areaRettangolo(4.0,2.0);
```

- Una volta eseguito questo codice la variabile *areaCalcolata* avrà valore 8.0.
- I valori che inseriamo al momento della chiamata del metodo sono detti *parametri attuali*
- Possiamo pensare che la macchina virtuale durante l'esecuzione del codice della chiamata al metodo salti letteralmente all'inizio della dichiarazione del metodo, inizializzi le variabili (formali) ***base*** ed ***altezza*** con i valori (attuali) 4.0 e2.0, esegua quindi il corpo del metodo fino a quando non incontra return, a quel punto prenda il valore dell'argomento di return e lo utilizzi per assegnare il valore alla variabile areaCalcolata.

METODI: SIGNATURE

- In Java un metodo è univocamente determinato dal suo identificatore e dalla *lista dei tipi dei parametri* che riceve in input che definisce la **signature del metodo** (la firma) quindi siamo liberi di ridefinire il medesimo metodo più volte a patto che ogni definizione abbia una lista di parametri diversa (tipi dei parametri, i nomi non contano, non conta nemmeno il tipo di ritorno).
- In questo modo possiamo sovraccaricare un identificatore di metodo con diverse definizioni ed effettuare il cosiddetto **overloading**.
- sarà il compilatore a scegliere al momento della chiamata quale metodo utilizzare sulla base del tipo (e del numero) degli argomenti attuali passati.

METODI: VISIBILITÀ

- I qualificatori del metodo **public, private** e **protected** hanno il medesimo significato fornito per le variabili e determinano quale parte del codice possa utilizzare quel metodo:
  - **public:** visibile dovunque,
  - **private:** solo alla classe,
  - **protected:** solo dalle classi che stanno nel medesimo package di quella in cui è definito il metodo e dalle classi da essa derivate,
  - **default:** implicherà visibilità alle classi nel medesimo package.
- Il qualificatore **final** serve per rendere un metodo non ridefinibile dalle sottoclassi, se un metodo viene contrassegnato come **final** le sottoclassi lo potranno utilizzare e lo erediteranno (quindi lo avranno disponibile) ma non potranno modificarlo (o come si dice comunemente: non potranno fare override del metodo).

METODI STATICI

L'uso di static ci si deve ricordare che ogni metodo appartiene ad una classe ed una classe è un "pacchetto" di dati e metodi.

Sappiamo che da una classe possiamo ottenere molteplici istanze e per ciascuna istanza si hanno variabili dai nomi identici ma dai valori distinti Se vogliamo che una variabile sia la medesima per tutte le istanze di una classe sappiamo che la dobbiamo invece definire come *static*.

- i **metodi non statici** sono associati ad ogni singola istanza di una classe e perciò il loro contesto di esecuzione (quindi l'insieme delle variabili cui possono accedere) è relativo all'istanza stessa: possono accedere e modificare le variabili dell'istanza e modificarne lo stato;

- i **metodi statici** non sono associati ad una istanza ma solo ad una classe. Quindi non potranno interagire con le variabili di istanza, ma solamente con quelle statiche.

USO DEI METODI

Tipo di metodo	Sintassi
Statico	NomeClasse.nomeMetodo(...)
Non statico (di istanza)	nomeIstanza.nomeMetodo(...)

Esempio possibile di metodo statico?

public static String calcolaCodFiscale(String nome, String cognome, Date nascita, String comune);

VARIABILI: CLASSIFICAZIONE

Si distinguono tre tipi di variabili:

- variabili locali
- variabili di istanza
- variabili di classe

VARIABILI LOCALI

La dichiarazione avviene all'interno di un metodo, create quando un metodo viene chiamato e scompaiono (cancellate dalla memoria) quando il metodo termina.

Ogni variabile dichiarata all'interno del metodo può essere utilizzata solamente all'interno del metodo

```
void add(long i) {

    long j = 1;

    j = j + i;

}
```

Nell'esecuzione di questo frammento di codice la macchina virtuale Java crea in memoria lo spazio per registrare le variabili locali i e j, cancellarle (liberando lo spazio in memoria) al termine del metodo

VARIABILE DI ISTANZA

Le variabili di istanza, sono dichiarate all'interno di una classe ma all'esterno di ogni metodo.

Hanno come scope l'intero corpo della classe in cui sono dichiarati, compresi i metodi della classe stessa, sono visibili all'interno di tutti i metodi della classe.

Può succedere che una variabile locale in un metodo (oppure il parametro di un metodo) abbia lo stesso nome (identificatore) di una variabile di istanza. In questo caso ha la precedenza la variabile più specifica, cioè la variabile locale o il parametro.

VARIABILI DI CLASSE

Le variabili di classe, dette anche *static field* o campi statici, sono variabili di istanza ma nella loro definizione viene usata la keyword 'static'.

```
static int var = 6;
```

Una variabile di classe è una variabile visibile da tutte le istanze di quell'oggetto ed il suo valore non cambia da istanza ad istanza, per questo appartiene trasversalmente a tutta la classe.

Mentre per le variabili di istanza viene allocata una nuova locazione di memoria per ogni istanza di una classe, per le variabili statiche esiste una unica locazione di memoria legata alla classe e non associata ad ogni singola istanza.

Una variabile di classe, statica, mantiene occupata la memoria e continua a mantenere il suo valore fino al termine del programma.

MODIFICATORI DI VISIBILITÀ

Le variabili possono essere qualificate per mezzo delle keywords public e private che ne determinano la visibilità all'esterno della classe in cui sono dichiarate.

- **private**: una variabile sarà visibile/accessibile solamente all'interno della classe;
- **public**: la variabile potrà essere utilizzata da qualsiasi parte del codice in cui ci sia una istanza della classe.
- **protected**: la variabile sarà accessibile da ogni altra classe che appartiene al medesimo package della classe che contiene la variabile e da ogni classe che ne deriva (la estende).
- Se non specifichiamo un qualificatore di visibilità, la variabile sarà lasciata con la visibilità di default

STATIC E STATIC FINAL

La keyword **final** per dichiarare una variabile che potrà essere inizializzata una sola volta, sia nella fase di dichiarazione o attraverso una successiva assegnazione.

In Java si definiscono **costanti** le variabili che vengono qualificate come **final e static**

LINGUAGGI OOP

Imparare la *programmazione orientata agli oggetti* richiede qualcosa di più della conoscenza del linguaggio di programmazione ma richiede lo schema mentale da assumere nell'affrontare la modellazione, l'analisi e l'implementazione della soluzione di un problema.

La programmazione orientata agli oggetti infatti ha una sua complessità che richiede sia la conoscenza di nozioni specifiche (alcune delle quali squisitamente teoriche), sia un opportuno metodo ed una disciplina.

GLI OGGETTI

Nella vita reale siamo abituati a classificare e a vedere oggetti dalle caratteristiche tangibili e riconoscibili, nel mondo OO invece, il concetto di oggetto si amplia e un oggetto può contenere elementi concreti, ma anche entità come processi

Gli oggetti hanno:

- **stato**, l'insieme delle variabili interne che ne definiscono le caratteristiche in un certo istante dell'esecuzione
- **comportamento**, le modalità di azione o di reazione di un oggetto, in termini di come il suo stato cambia e come i messaggi passano

INTERAZIONE DI OGGETTI

L'interazione tra diversi oggetti avviene attraverso lo **scambio di messaggi**

Un oggetto, detto *sender* del messaggio agisce su un altro oggetto, detto *recipient*, spedendogli uno dei messaggi che il ricevente è in grado di accettare.

Ovvero l'oggetto sender chiama uno dei metodi "esposti" dall'oggetto ricevente.

In Java intendiamo per "oggetto" l'istanza particolare di una certa classe, e esso può possedere/esporre alcuni metodi, quindi un oggetto può ricevere un certo messaggio se possiede un metodo che l'oggetto sender è in grado di chiamare con la opportuna visibilità.

CLASSI

Per ogni oggetto è necessario sapere qual è il set dei messaggi che è in grado di ricevere. Non possiamo infatti inviare messaggi ad un oggetto se essi non sono ammessi nell'insieme delle azioni previste.

Una classe è la descrizione/prototipo di un oggetto in cui vengono definiti tutti i messaggi che ciascuna istanza sarà in grado di ricevere.

Nella classe si devono definire:

- l'insieme delle variabili di stato (**attributi**) che ne rappresentano lo stato
- l'insieme dei **metodi** che la classe supporta (messaggi ricevibili)

ESEMPIO DI CLASSE

```
public class Conto {
    private double amount;
    private String owner;

    public Conto(String owner, double initialAmount) {
        this.owner = owner;
        this.amount = initialAmount;
    }

    public void versamento(double qty) {
        amount += qty;
    }

    public boolean prelievo(double qty) {
        if(amount < qty)
            return false;
        amount -= qty;
        return true;
    }

    public double getAmount() {
        return amount;
    }

    public String getOwner() {
        return owner;
    }
}
```

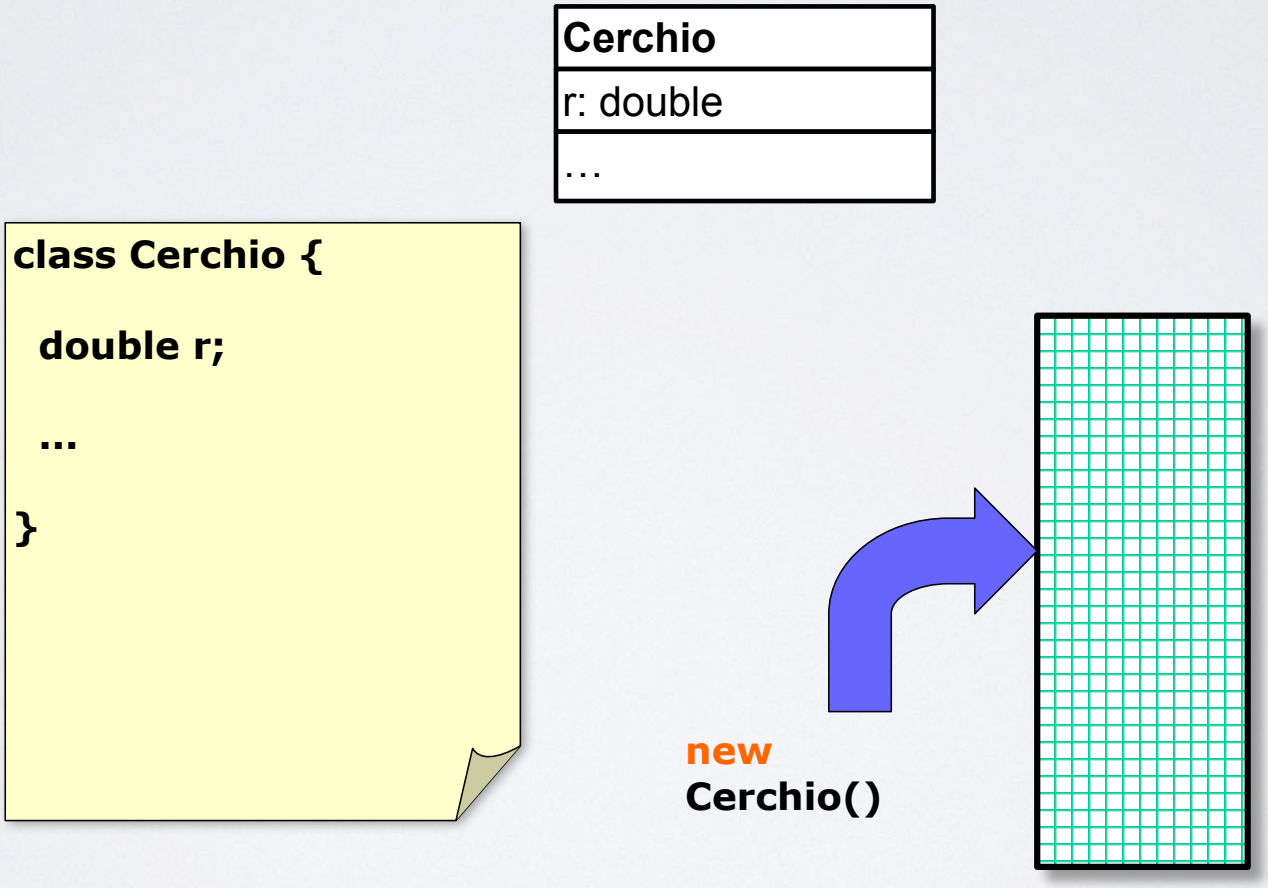
COSTRUTTORE

- Il costruttore è quel metodo di una classe il cui compito è proprio quello di **creare nuove istanze**, oltre ad essere il punto del programma in cui un nuovo elemento (quindi una nuova identità) viene creato ed è reso disponibile per l'interazione con il resto del sistema.
- Il costruttore è spesso usato per effettuare le inizializzazioni dello stato delle nuove istanze.
- Possono esserci molteplici costruttori per una medesima classe (ognuno con parametri di diversi) e ne esiste sempre almeno uno.
- Se per una classe non viene specificato alcun costruttore, il compilatore ne sintetizza automaticamente uno senza argomenti: *costruttore default.*

OGGETTO ISTANZA DI CLASSI

- Data una classe, è possibile costruire uno o più oggetti
  - Gli oggetti vengono detti "istanze" della classe
  - In Java si utilizza la notazione ***new NomeClasse ( );***
- Ogni oggetto "vive" all'interno della memoria del calcolatore
  - Qui viene memorizzato il suo stato
  - Oggetti differenti occupano posizioni differenti

OGGETTO ISTANZA DI CLASSI



CLASSI E OGGETTI

Un nuovo oggetto si istanzia con il comando new: Car myCar = new Car();

Una classe può definire un metodo costruttore (sempre public), o anche più di uno, per la sua inizializzazione:

```
public class Car {

 private String model;

 public Car(String model) { this.model = model; }

}
```

In tal caso l'oggetto deve essere istanziato con i paramatri necessari:

```
Car myCar = new Car("Fiat Duna");
```

CLASSI E OGGETTI

Per la precisione, la variabile myCar è un riferimento all'oggetto che è stato istanziato (e che occupa memoria fisica). La seguente istruzione:

```
Car yourCar = myCar;
```

Non crea una copia dell'oggetto a cui myCar fa riferimento!

L'oggetto è sempre uno solo, solo che ora ci sono due variabili che vi fanno riferimento!

CLASSI E OGGETTI

Se non uso il comando new ma scrivo semplicemente: Car myCar;

il compilatore segnala errore: *The local variable myCar may not have been initialized*

A volte, anziché usare new o assegnare alla variabile un riferimento valido, si dichiara esplicitamente la variabile null, ad esempio perché l'oggetto a cui il riferimento deve puntare viene istanziato all'interno di un successivo blocco nel contesto di una operazione di controllo di flusso.

```
Car myCar = null;
if (carType.equals("sport")) {
    myCar = new SportCar();
    ....
}
else if (carType.equals("suv"))
    myCar = new SuvCar();
myCar.run();
```

La dichiarazione va fuori dai blocchi, altrimenti la variabile risulta visibile solo all'interno del blocco in cui viene fatta!

È il concetto di SCOPE di una variabile.

Vale per tutti i tipi di variabile!

RIFERIMENTI

- Si opera su un oggetto attraverso un riferimento e indica la posizione in memoria occupata dall'oggetto
- All'atto della costruzione, l'operatore **new** si effettuano le seguenti operazioni:
  - Alloca un **blocco** di memoria sufficiente a contenere l'oggetto
  - Invoca il **costruttore**, determinandone la corretta inizializzazione
  - Restituisce il **riferimento** (indirizzo) del blocco inizializzato

ACCESSIBILITÀ

- Un oggetto è accessibile fino a che ne esiste un riferimento, nel momento in cui non esistano più riferimenti, l'oggetto può essere eliminato, rilasciando la memoria che occupa
- I riferimenti sono memorizzati in variabili e attributi
  - Si cancellano quando la variabile cessa di esistere (fine del blocco)
  - Oppure assegnando esplicitamente il valore **null**

CONTEGGIO DEI RIFERIMENTI

- All'interno di ogni oggetto, Java mantiene un contatore nascosto
  - Indica il numero di riferimenti esistenti a quello specifico oggetto
  - Quando il suo valore scende a 0, indica che l'oggetto può essere eliminato, rilasciando la memoria che occupa
- Un particolare sottosistema, il garbage collector, si occupa, periodicamente, di riciclare la memoria degli oggetti eliminati
  - Viene eseguito automaticamente dalla macchina virtuale Java

PROGRAMMAZIONE OBJECT ORIENTED

- Classi: definizione attributi e metodi di classe con costruttore
- Introduzione alla programmazione ad oggetti e progettazione (OO)
- Ereditarietà e sue applicazioni
- Classi e metodi abstract
- Interfacce e loro impiego
- Polimorfismo e sue applicazioni pratiche

Dott. Antonio Giovanni Lezzi Afol Milano

PROGRAMMAZIONE OO

- La programmazione orientata agli oggetti (Object Oriented Programming, OOP) è un paradigma di programmazione, in cui un programma viene visto come un insieme di oggetti che interagiscono tra di loro.
- Nei linguaggi OOP esiste un nuovo tipo di dato, la classe. Questo tipo di dato serve appunto a modellare un insieme di oggetti dello stesso tipo.
- In generale, un oggetto è caratterizzato da un insieme di attributi e da un insieme di funzionalità.

INTRODUZIONE PROGRAMMAZIONE AD OGGETTI

- Il modo di pensare come programmare un applicazione è fondamentale.
- In programmazione imperativa ci mettiamo davanti il progetto e ci chiediamo cosa fare, fissando l'attenzione sull'obiettivo da raggiungere e di come raggiungerlo.
- In programmazione ad oggetti, invece, dobbiamo cercare di non personalizzare l'oggetto ma continueremo a pensare l'oggetto così come è.
- Penseremo ad un oggetto astratto, cercando di individuare quali sono le caratteristiche dell'oggetto e le sue potenzialità.
- Gli elementi caratteristici sono chiamati Attributi dell'oggetto, le potenzialità sono invece chiamate Metodi.

ERRORI COMUNI

- Un errore che spesso si sente è l'utilizzo di Classe e Oggetto come se fossero la stessa cosa: un Oggetto è la rappresentazione reale della problematica presa in considerazione, la Classe invece è la rappresentazione astratta dell'Oggetto.
- Per esempio, se consideriamo una penna, essa nella realtà rappresenta il nostro Oggetto vero e proprio, così come è fatto: sappiamo che è una penna, il colore, il tipo, ecc...
- La Classe che rappresenterà la penna sarà, per definizione, l'astrazione dell'Oggetto. Qui ci possiamo chiedere "E cosa cambia? Non è la stessa cosa?", la risposta è parzialmente negativa perché, attraverso questo processo di astrazione, noi andremo a determinare anche altri dettagli supplementari dell'Oggetto: se scrive o meno, livello di inchiostro, ecc.

PROGRAMMI STRUTTURATI CONTRO OBJECT ORIENTED

La programmazione ad oggetti ha un diverso approccio alla decomposizione dei problemi:

- Con l'*approccio strutturale* ci si concentra sulla scomposizione di algoritmi in procedure.
- Nell'*approccio "a oggetti"* ci si focalizza sull'interazione di elementi (oggetti) che comunicano (scambiano messaggi) tra loro.

OBJECT ORIENTED PROGRAMMING

È un metodo di implementazione in cui i programmi sono organizzati attraverso un insieme di oggetti, ognuno dei quali è un'istanza di una classe. Queste classi sono tutte parte di una gerarchia di entità unite fra di loro da una relazione di ereditarietà.

Ci sono tre parti particolarmente importanti:

1. OOP utilizza un insieme di **oggetti** (non algoritmi, e gli oggetti sono la parte fondamentale della costruzione logica);

2. ogni oggetto è istanza di una **classe**;

3. ogni classe è legata alle altre attraverso una relazione detta **eredità**.

CARATTERISTICHE OO

Si individuano solitamente nello stile Object Oriented i seguenti elementi:

- Astrazione
- Incapsulamento
- Gerarchia
- Modularità

ASTRAZIONE

L'astrazione ci consente di evidenziare le caratteristiche fondamentali di un oggetto e di classificarlo simile ad altri dello stesso tipo e distinto da tutti gli altri tipi e quindi ci consente di tracciare confini concettuali ben definiti per descriverlo all'interno di un certo **contesto di osservazione** che ci interessa. Questo può cambiare da contesto a contesto!

È importante osservare che l'astrazione va utilizzata come strumento che permette di **focalizzare l'attenzione su una visione esterna di un oggetto**, in modo da separare quella che è l'implementazione di un comportamento dal suo ruolo nella dinamica globale di un determinato processo.

INCAPSULAMENTO

I concetti di astrazione e di incapsulamento sono complementari:

- l'astrazione focalizza l'attenzione sul comportamento e le caratteristiche osservabili di un oggetto
- l'incapsulamento si concentra sull'implementazione che riesce a riprodurre queste caratteristiche.

L'incapsulamento è molto spesso ottenuto attraverso l'**aggregazione di informazioni**, che è il processo di nascondere tutte le informazioni private di un oggetto che non contribuiscono a definire quelle che sono le sue caratteristiche essenziali nell'interazione con le altre entità del sistema in esame.

INCAPSULAMENTO

Tipicamente la struttura di un oggetto viene mantenuta nascosta, così come l'implementazione dei suoi metodi consentendoci di creare delle vere e proprie scatole chiuse fra i diversi tipi di astrazioni che possono essere definite per un oggetto.

L'incapsulamento è il processo di suddivisione degli elementi di un'astrazione che ne costituiscono la struttura e il comportamento; incapsulamento serve a separare l'interfaccia costruita per un certo tipo di astrazione e la sua attuazione.

GERARCHIA

- Nella maggior parte dei problemi che si affrontano non esiste una unica astrazione da tenere in considerazione ma spesso ne esistono molteplici e la possibilità di organizzarle in gerarchie è di vitale importanza per una modellazione efficace.
- L'incapsulamento ci aiuta nella gestione di questa complessità cercando di mantenere nascosta all'utente finale la nostra astrazione e la modularità ci da il modo per gestire le relazioni logiche dell'astrazione.
- Le gerarchie che comunemente si descrivono nella programmazione sono quelle che potremmo definire strutturali (un determinato oggetto "è un" cioè "appartiene alla tal categoria") oppure quelli comportamentali "si comporta come un".

MODULARITÀ

- Il significato di modularità può essere visto come l'azione di partizionare un programma / problema in componenti che riescano a ridurne il grado di complessità.
- Nella modellazione di sistemi complessi l'astrazione in entità (e gerarchie), l'incapsulamento non bastano per rendere trattabili problemi ma è necessario ricorrere anche alla modularizzazione
- La modularizzazione è l'individuazione di gruppi di entità (classi) affini per funzionalità, area di utilizzo, comportamento e conseguente divisione del problema in sottoproblemi più facilmente affrontabili.
- Questa partizione in moduli consente anche di creare un certo numero elementi ben definiti e separati all'interno del programma stesso, ma che sia possibile connettere fra di loro per orchestrare una soluzione globale al problema in esame.

DOMANDE DA PORSI

- Quali sono le entità in gioco nel nostro dominio: queste diventeranno le classi nel linguaggio di programmazione a oggetti;
- Come si relazionano tra loro le varie entità;
- Come sono distribuite le responsabilità tra le entità in gioco: chi fa cosa e come avviene la collaborazione per arrivare a raggiungere l'obiettivo prefissato;
- Architettura: organizzazione dei progetti, individuazione di componenti riusabili, di servizi generici e così via

RIUSARE IL SOFTWARE

- A volte si incontrano classi con funzionalità simili
  - In quanto sottendono concetti semanticamente "vicini"
  - Una mountain bike assomiglia ad una bicicletta tradizionale
- È possibile creare classi disgiunte replicando le porzione di stato/comportamento condivise
  - L'approccio "Taglia&Incolla", però, non è una strategia vincente
  - Difficoltà di manutenzione correttiva e perfettiva
- Meglio "specializzare" codice funzionante
  - Sostituendo il minimo necessario

PROGRAMMAZIONE OBJECT ORIENTED

- Classi: definizione attributi e metodi di classe con costruttore
- Introduzione alla programmazione ad oggetti e progettazione (OO)
- Ereditarietà e sue applicazioni
- Classi e metodi abstract
- Interfacce e loro impiego
- Polimorfismo e sue applicazioni pratiche

Dott. Antonio Giovanni Lezzi Afol Milano

EREDITARIETÀ

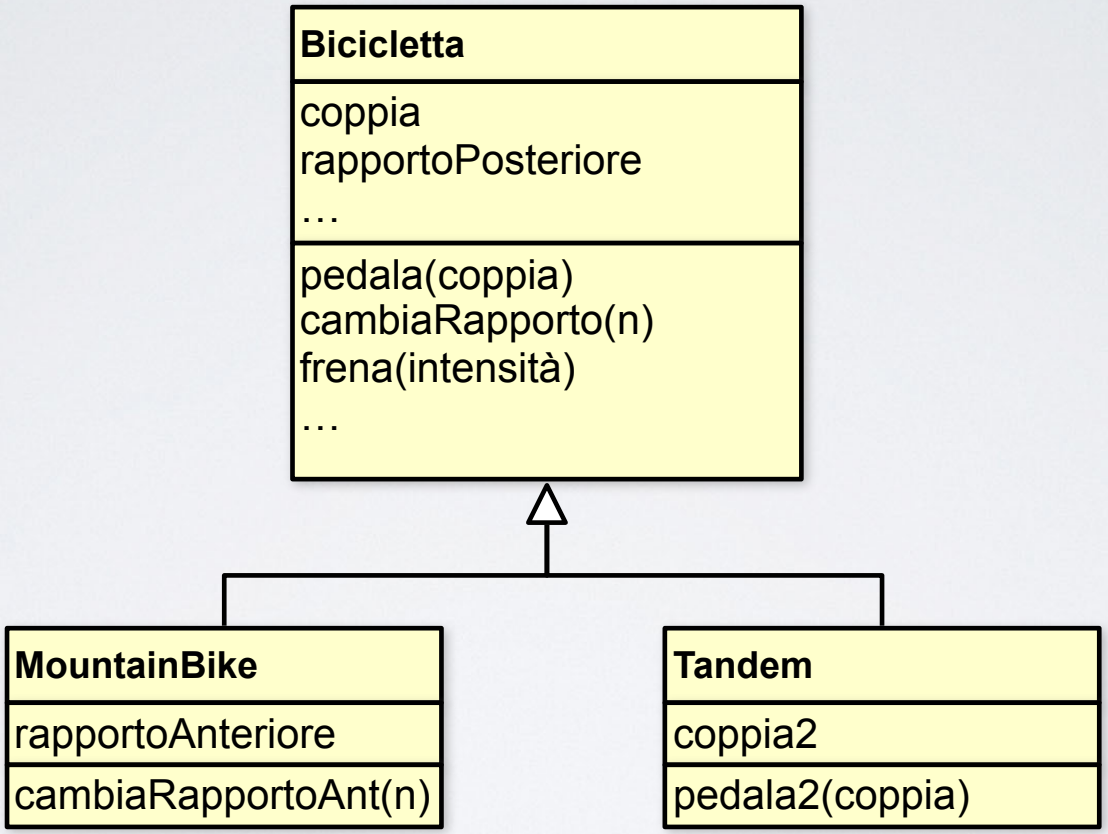
Si dice che una classe A è una sottoclasse di B (e analogamente che *B* è *una superclasse di A*) quando:

- *A* eredita da *B* sia il suo stato che il suo behavior (comportamento)
- un'istanza della classe *A* è utilizzabile in ogni parte del codice in cui sia possibile utilizzare una istanza della classe *B*

EREDITARIETÀ

- Meccanismo per definire una nuova classe (classe derivata) come specializzazione di un'altra (classe base)
  - La classe base modella un concetto generico
  - La classe derivata modella un concetto più specifico
- La classe derivata:
  - Dispone di tutte le funzionalità (attributi e metodi) di quella base
  - Può aggiungere funzionalità proprie
  - Può ridefinirne il funzionamento di metodi esistenti (polimorfismo)

EREDITARIETÀ



TIPI DI EREDITARIETÀ

- Ogni classe definisce un tipo:
  - Un oggetto, istanza di una sotto-classe, è formalmente compatibile con il tipo della classe base
  - Il contrario non è vero!
- Esempio: Un'automobile è un veicolo ma veicolo non è (necessariamente) un'automobile
- La compatibilità diviene effettiva se imetodi ridefiniti nella sotto-classe rispettano la semantica della superclasse
- L'ereditarietà gode delle proprietà transitiva: Un tandem è un veicolo (poiché è una bicicletta, che a sua volta è un veicolo)

VANTAGGI EREDITARIETÀ

- Evitare la duplicazione di codice
- Permettere il riuso di funzionalità
- Semplificare la costruzione di nuove classi
- Facilitare la manutenzione
- Garantire la consistenza delle interfacce

EREDITARIETÀ IN JAVA

- Si definisce una classe derivata attraverso la parola chiave "extends" seguita dal nome della classe base
- Gli oggetti della classe derivata sono, a tutti gli effetti, estensioni della classe base, anche nella loro rappresentazione in memoria
- Non esiste l'ereditarietà multipla! NuovaClasse può estendere solo una classe esistente.

EREDITARIETÀ IN JAVA

Se si deve realizzare una nuova classe ed è già disponibile una classe che rappresenta un concetto più generale, la nuova classe può essere derivata dalla classe esistente (superclasse), di cui eredita attributi e metodi.

```
public class NuovaClasse extends Superclasse {

 nuovi campi di esemplare

 nuovi metodi

}
```

Non esiste l'ereditarietà multipla! NuovaClasse può estendere solo una classe esistente.

PROGRAMMAZIONE OBJECT ORIENTED

- Classi: definizione attributi e metodi di classe con costruttore
- Introduzione alla programmazione ad oggetti e progettazione (OO)
- Ereditarietà e sue applicazioni
- Classi e metodi abstract
- Interfacce e loro impiego
- Polimorfismo e sue applicazioni pratiche

Dott. Antonio Giovanni Lezzi Afol Milano

EREDITARIETÀ ED ABSTRACT

- Una classe dichiarata abstract deve essere per forza estesa da un'altra classe.
- Se un metodo è dichiarato abstract, non può essere usato. Solo le classi che estendendo la prima ne implementano tale metodo, lo rendono disponibile.

```
public abstract class NomeClasse {  // non istanziabile

    public void metodo1() { … }  // non utilizzabile

    private int metodo3() {

    }

}
```

PROGRAMMAZIONE OBJECT ORIENTED

- Classi: definizione attributi e metodi di classe con costruttore
- Introduzione alla programmazione ad oggetti e progettazione (OO)
- Ereditarietà e sue applicazioni
- Classi e metodi abstract
- Interfacce e loro impiego
- Polimorfismo e sue applicazioni pratiche

Dott. Antonio Giovanni Lezzi Afol Milano

EREDITARIETÀ ED INTERFACCE

Per aggirare il limite sull'ereditarietà multipla, si usano le interfacce.

Una interfaccia è una classe senza variabili membro, senza costruttore e i cui metodi sono solo dichiarati (e sono tutti public, è sottinteso)

```
public interface
    Shape {
    double
    getArea();
}

public interface
    Colored {
    String
    getColor();
}
```

```
public class ColoredCircle implements Shape, Colored {
    double radium = 0;
    String color = null;
    public ColoredCircle(double radium, String color) {
        this.radium = radium;
        this.color = color;
    }
    public double getArea() {
                return Math.PI*radium*radium; }
    public String getColor() { return color; }
}
```

EREDITARIETÀ ED INTERFACCE

Non è possibile istanziare un oggetto da una interface. Se Circle è una interfaccia, la seguente istruzione è sbagliata:

```
Circle myCircle = new Circle();  //non si può fare

Circle myCircle = new ColoredCircle("blue", 10); //questo si può fare
```

Questo ha importanti implicazioni (in relazione al polimorfismo..)!

- Una classe può implementare più interfacce
- Più classi possono implementare una stessa interfaccia.
- Una classe può estendere una sola classe e contemporaneamente implementare una o più interfacce.

INTERFACCIA VS ASTRATTA

- Si usa **un'interfaccia** quando si deve definire il tipo di una classe identificando le operazioni offerte e astraendo da come sono implementate. Il polimorfismo viene sfruttato per dare comportamenti diversi alla stessa operazione.

- Si usa una **classe astratta** quando in un design si vogliono fattorizzare alcuni comportamenti a livello di super classe, mentre altri comportamenti devono essere ridefiniti. Questa super non è comunque sufficiente a caratterizzare entità concrete per cui istanze di tale classe non devono poter essere create.

PROGRAMMAZIONE OBJECT ORIENTED

- Classi: definizione attributi e metodi di classe con costruttore
- Introduzione alla programmazione ad oggetti e progettazione (OO)
- Ereditarietà e sue applicazioni
- Classi e metodi abstract
- Interfacce e loro impiego
- Polimorfismo e sue applicazioni pratiche

Dott. Antonio Giovanni Lezzi Afol Milano

POLIMORFISMO

- In Java, una interfaccia dichiara un insieme di metodi (tutti pubblici, anche se non viene dichiarato esplicitamente) e le loro signature.
- Diversamente da una classe, un'interfaccia non fornisce alcuna implementazione.
- Una classe che è dichiarata come implementazione di una interfaccia deve implementare tutti i metodi di quest'ultima.
- Quando più classi realizzano la medesima interfaccia, ciascuna classe può realizzare a modo suo i metodi propri dell'interfaccia.
- Una classe che estende un'altra classe può ridefinirne i metodi (**override**). Anche questo è polimorfismo.

POLIMORFISMO

```
public interface Measurable {

        double getMeasure();

}

public class BankAccount implements Measurable {

        private double balance;

        public double getMeasure() { return balance;}

}

public class Coin implements Measurable {

        private double value;

        public double getMeasure() {  return value;}

};
```

POLIMORFISMO

È stato detto che è possibile dichiarare una variabile di tipo interfaccia, purché faccia riferimento a un oggetto di una classe che implementa tale interfaccia:

```
Measurable meas = new BankAccount();
```

Questa possibilità offerta da Java ha un motivo profondo: rende possibile definire metodi che accettano parametri di tipo interfaccia, e questi metodi accettano istanze di tutte le classi che implementano tale interfaccia!

```
public void register(Measurable meas) { .. }  // metodo di una qualche classe
```

Quando invochiamo register() possiamo passargli un oggetto di tipo BankAccount, oppure di tipo Coin! Se modifichiamo qualcosa di queste due classi (ad es. il nome) non dobbiamo modificare il metodo register().

POLIMORFISMO

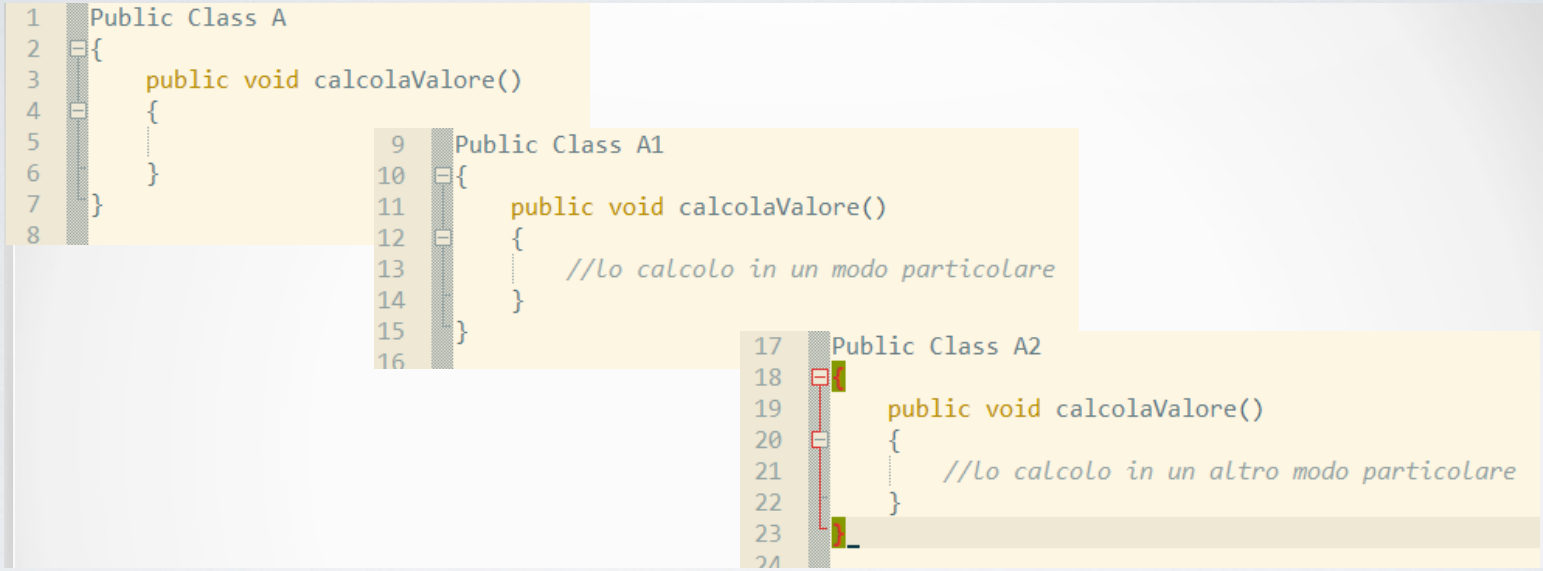
Una classe che estende un'altra classe può ridefinire i metodi.

Anche questo è polimorfismo.

```
public class Animale {
    public void interroga() {
            System.out.println("Grunt");
    }
}

public class Ghepardo extends Animale {
    public void interroga() {
            System.out.println("Groar!");
    }
    public void salta() {
            System.out.println("hop!");
    }
}
```

CLASSI: POLIMORFISMO



```
Public Class A
{
    public void calcolaValore()
    {

    }
}
```

```
Public Class A1
{
    public void calcolaValore()
    {
        //lo calcolo in un modo particolare
    }
}
```

```
Public Class A2
{
    public void calcolaValore()
    {
        //lo calcolo in un altro modo particolare
    }
}
```

- Estendere una classe padre A in un più classi figli A1 e A2, consente di poter richiamare in runtime i metodi condivisi senza sapere a priori se sto usando la classe A1 o A2.

ESERCIZIO

Si costruisca un sistema contenente informazioni relative ad una azienda il cui scopo è quello di profilare i propri collaboratori: dipendenti e consulenti (con le relative caratteristiche che ogni profilo possono avere), effettuare controllo accessi, gestire le commesse da assegnare al team e la gestione di un sistema di fatturazione.