

tree **school**

Corso Java Backend
Modulo 4

Programmazione Object Oriented

Melvin Massotti





Di cosa parliamo

Object Oriented Programming

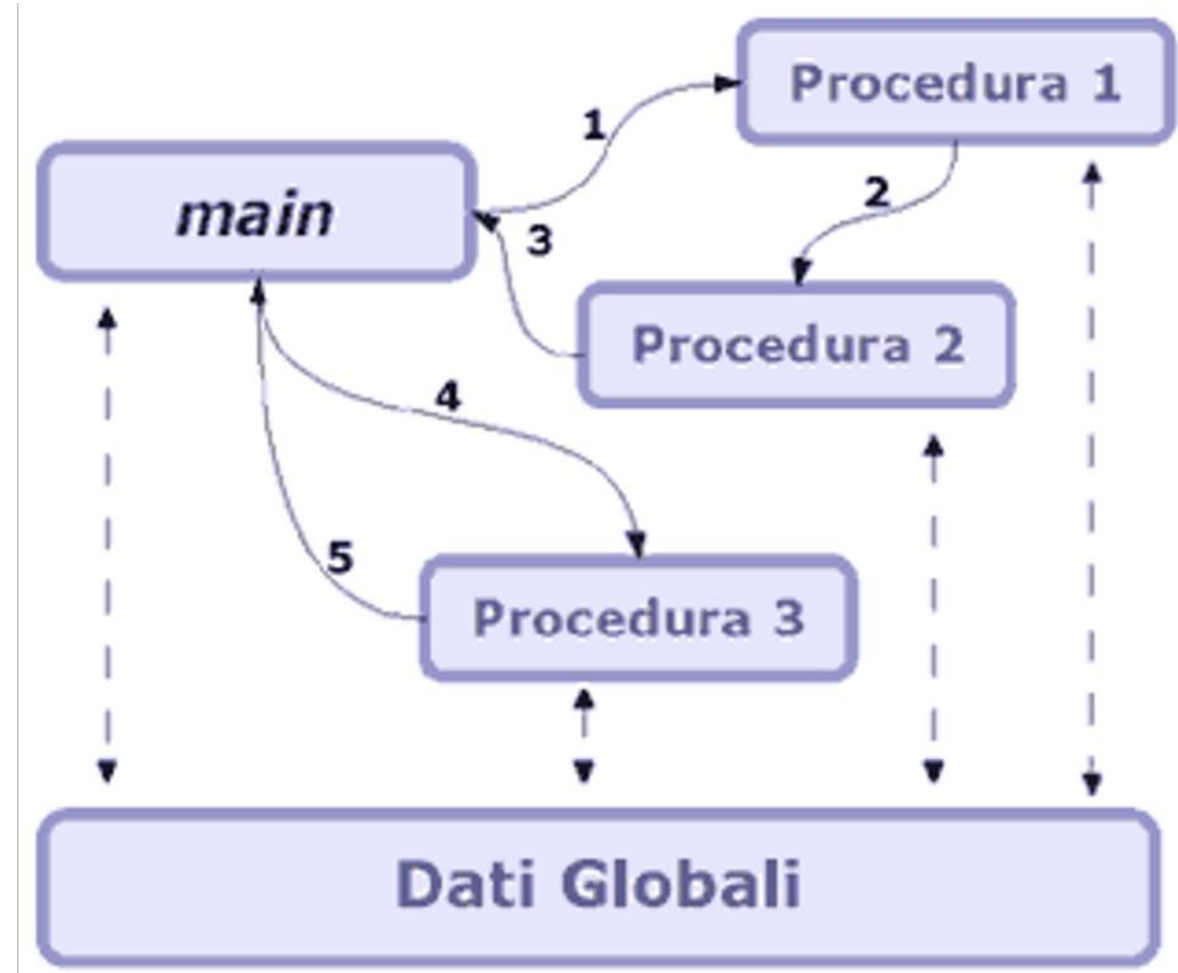
- Paradigma Object Oriented
- Principi dell'OOP
- Classi ed Oggetti
- Struttura di una classe
- Lavorare con gli oggetti
 - Dichiarazione
 - Proprietà e tipi
 - Metodi, parametri, tipi di ritorno
 - Overloading
 - Istanziamento
 - Accesso alle priorità
 - Getter e Setter
 - Reference vs Object vs Istanza vs Classe
 - Keyword «static»
 - Keyword «final»
 - Enumerazioni

Object Oriented Programming



Prima della OOP

Il paradigma di programmazione più comune era quello della programmazione procedurale, dove i pezzi di codice ripetuti vengono raggruppati in procedure che successivamente vengono richiamate ogni volta che se ne presenti l'esigenza. Il programma è costituito da un unico file.



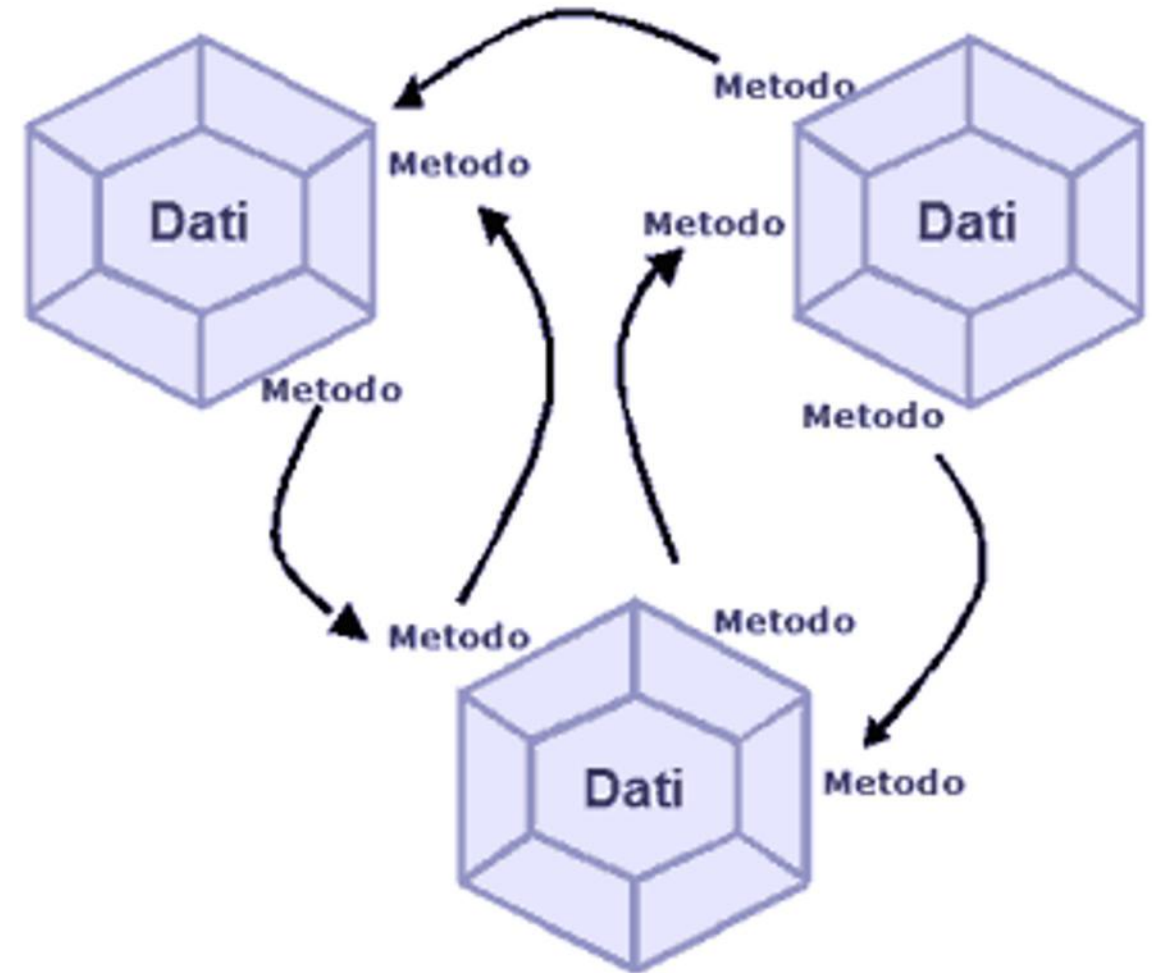
Prima della OOP

Cosa non va con la programmazione procedurale?

- Poco riutilizzo del codice
- Le applicazioni sono difficili da espandere e mantenere
- Bisogna stare attenti ai dati globali
- Non si possono nascondere le informazioni e quindi qualsiasi dato è accessibile a qualsiasi parte di codice.
- Difficile da progettare poiché il focus è sulle azioni e non sui dati.
- Livello di astrazione medio/basso.

L'arrivo della OOP

Da queste esigenze nasce un nuovo modo di progettare il software, orientato ai dati piuttosto che alle azioni, puntando su una maggiore modularità e aumentando il livello di astrazione del software.



Qualche cenno di storia

Anni 60: Nascita dei primi linguaggi di programmazione, Simula 1 e Simula 67

Anni 70: Nasce SmallTalk dai laboratori della Xerox, il primo linguaggio orientato agli oggetti, il suo successo convince la comunità di Lisp ad aggiungere feature OOP al linguaggio.

Anni 80: Molti altri linguaggi come il COBOL, Ada, Fortran e BASIC aggiungono feature della programmazione ad oggetti. Nascono nuovi linguaggi OOP come Objective-C.

Anni 90: La programmazione ad oggetti diventa il paradigma di programmazione più usato al mondo, nascono linguaggi come il C++ e Java che ancora oggi sono tra i più utilizzati a livello globale.

Oggi: Nonostante la nascita e l'ascesa di nuovi paradigmi di programmazione la OOP è ancora largamente diffusa e mantiene solido il suo primato.

L'obiettivo di un linguaggio di programmazione

- Rendere la vita più semplice al programmatore
- Semplificare i processi di sviluppo
 - Progettazione
 - Coding
 - Testing
 - Manutenzione

Le idee alla base della OOP

- Dividere il programma in classi permette di lavorare più facilmente grazie all'uso di componenti più piccole
- Qualsiasi concetto può essere rappresentato tramite un oggetto, definendone la classe
- Possono esistere più istanze di una singola classe
- Ogni oggetto, quindi ogni istanza di una classe, ha il proprio stato, descritto dai valori assegnati ai suoi attributi

Il Paradigma Object Oriented



Il Paradigma Object Oriented

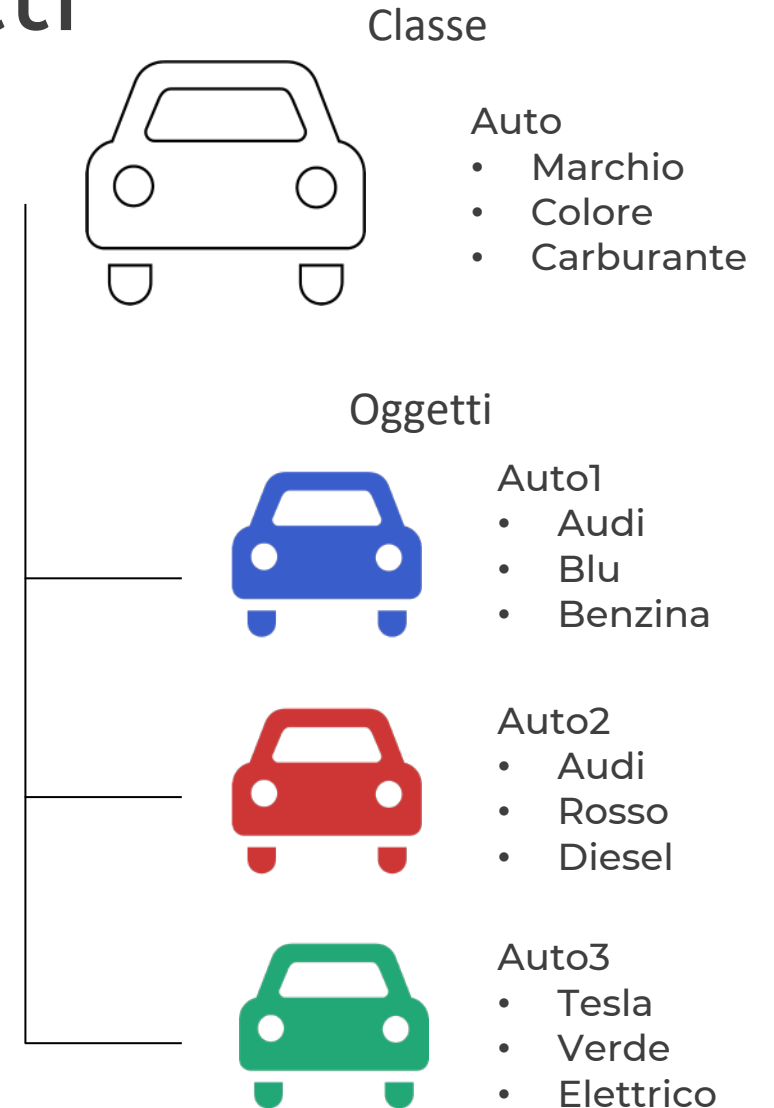


Classi e Oggetti

- Le classi, e quindi gli oggetti, sono i mattoncini della progettazione nella OOP
- Le classi modellano la realtà rappresentando
 - Oggetti reali
 - Entità software
 - Concetti astratti
- Una classe è quindi il prototipo astratto di un oggetto, che lo definisce mediante
 - Campi (o attributi)
 - Metodi

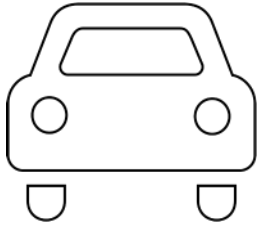
Classi e Oggetti

- Un oggetto è un'istanza di una classe
- Un programma può creare ed usare una o più istanze di una classe
- Ogni oggetto può presentare valori diversi per i suoi attributi



Classi e Oggetti

Classe: Automobile



Campi (o attributi)

- String marchio
- Color colore
- String carburante
- int numeroPasseggeri

Metodi

- aggiungiPasseggero()
- qualeCarburante()

Oggetto: Una certa automobile



Campi (o attributi)

- String marchio = "Audi"
- Color colore = Colors.BLUE
- String carburante = "Benzina"
- int numeroPasseggeri = 5

Metodi


- aggiungiPasseggero()
- qualeCarburante()

Classi e Oggetti


Esercizio


Ducati Multistrada 1200 ENDURO Pack TOURING


Enduro | Contatta il tuo Personal CAR Shopper!



1/16

 Salva


 Condividi

 Stampa

€ 15.000,-

Prezzo finale¹

Pubblicità - Esempio di finanziamento

 [Imp. mensile da](#) **€ 290,-²**

[Dettagli del Finanziamento](#) ▾

14.666 km Usato

07/2019 1 proprietario

118 kW 160 CV Manuale, Benzina

Castellammare Di Stabia - Napoli - NA, Annuncio di rivenditore


Studio81 ★★★★★ (48)

8i

Contatta venditore

+39 081 - 19288101

Acquisto veloce



Classi e Oggetti

Esercizio

Moto

- Marchio
- Colore
- anno
- mese
- kmPercorsi
- Prezzo
- telVenditore

Moto

- pilota
- marchio
- maxVel
- numero
- maneggevolezza

Struttura di una classe

Dichiarazione

```
Automobile.java x
1 public class Automobile {
2
3 }
4
```

```
Automobile.java x
1 class Automobile {
2
3 }
4
5
```

Struttura di una classe

Campi

```
public class Automobile {  
  
    /**  
        {public | private | protected} {static} {final} {tipo} {nome};  
    **/  
    public static final int ruote = 4;  
    private String marchio;  
  
}
```

```
double test = Math.PI;
```

Struttura di una classe

Costruttore

```
public class Automobile {  
  
    public static final int ruote = 4;  
    private String marchio;  
  
    /**  
     {public | private | protected} {nome classe }({parametro1, parametro2, etc}){  
     |   campo1 = parametro1;  
     |   campo2 = parametro2;  
     |}  
    */  
  
    public Automobile(){  
        marchio = "Audi";  
    }  
  
    public Automobile(String marchio) {  
        this.marchio = marchio;  
    }  
}
```

Struttura di una classe

Metodi

```
public class Automobile {  
  
    public static final int ruote = 4;  
    private String marchio;  
  
    public Automobile(String marchio) {  
        this.marchio = marchio;  
    }  
  
    /**  
     {public | private | protected} {static} {tipo} {nome} ({parametri}){  
     istruzioni...  
     }  
    */  
  
    public void setMarchio(String marchio){  
        this.marchio = marchio;  
    }  
  
    public String getMarchio(){  
        return marchio;  
    }  
  
    public void faiRifornimento(){  
        //  
    }  
}
```

Struttura di una classe

Metodi

Metodi statici

Anche un metodo può essere statico e, come i campi statici, è accessibile senza dover istanziare un oggetto della classe

Si usano ad esempio per metodi di supporto (es. classe Math)

Non possono usare campi d'istanza, perché questo non renderebbe più il loro comportamento unico, quindi prendono i dati solo dai parametri o dai campi statici

Un metodo che non usa campi d'istanza probabilmente dovrebbe essere statico, non è un errore per il compilatore ma potrebbe essere un errore logico di progettazione, ed in ogni caso renderebbe il comportamento del codice più chiaro

```
public class Utilities {  
  
    public static double moltiplica(int a, int b){  
        return a*b;  
    }  
  
    public void doSomething(){  
        double result = moltiplica( a: 3, b: 4);  
    }  
}
```

```
public static void main(String[] args) {  
  
    double result = Utilities.moltiplica( a: 5, b: 6);  
}
```

Struttura di una classe

Metodi

Overloading

L'overloading è un meccanismo che permette di chiamare più metodi nella stessa classe con lo stesso nome purché ogni metodo abbia una **firma** differente (escludendo il nome)

La firma di un metodo è composta dal nome del metodo, dal tipo dei parametri e dall'ordine dei parametri

<nome>(<tipoParam1>, <tipoParam2>....)

Ad esempio il metodo

```
public static int somma(int x, int y)
```

ha come firma

```
somma(int, int)
```

```
public class Utilities {  
  
    public static double moltiplica(int a, int b){  
        return a*b;  
    }  
  
    public static double moltiplica(double a, double b){  
        return a*b;  
    }  
  
    public void doSomething(){  
  
        int intero1 = 3;  
        int intero2 = 4;  
  
        double result = moltiplica(intero1, intero2);  
  
        double double1 = 3.0;  
        double double2 = 4d;  
  
        double result2 = moltiplica(double1, double2);  
  
    }  
}
```

Utilizzo di una classe

```
public class Orologio {  
  
    public int ora;  
    public int minuto;  
  
    public String getOrario() {  
        return ora + ":" + minuto;  
    }  
  
}
```

```
public static void main(String[] args) {  
  
    Orologio orologio = new Orologio();  
    orologio.ora = 19;  
    orologio.minuto = 20;  
    orologio.getOrario(); // 19:20  
}
```

Con i campi pubblici posso accedere direttamente al campo

Utilizzo di una classe

```
public class Orologio {  
  
    private int ora;  
    private int minuto;  
  
    public Orologio(int ora, int minuto) {  
        this.ora = ora;  
        this.minuto = minuto;  
    }  
  
    public int getOra() {  
        return ora;  
    }  
  
    public void setOra(int ora) {  
        this.ora = ora;  
    }  
  
    public int getMinuto() {  
        return minuto;  
    }  
  
    public void setMinuto(int minuto) {  
        this.minuto = minuto;  
    }  
  
    public String getOrario(){  
        return ora+":"+minuto;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Orologio orologio = new Orologio( ora: 19, minuto: 20);  
    orologio.getOrario(); // 19:20  
  
    orologio.setOra(15);  
    orologio.setMinuto(10);  
    orologio.getOrario(); // 15:10  
}
```

Con i campi privati devo usare getter e/o setter

Utilizzo di una classe

```
public class Orologio {  
  
    private int ora;  
    private int minuto;  
  
    public Orologio(int ora, int minuto) {  
        this.ora = ora;  
        this.minuto = minuto;  
    }  
  
    public int getOra() {  
        return ora;  
    }  
  
    public void setOra(int ora) {  
        this.ora = ora;  
    }  
  
    public int getMinuto() {  
        return minuto;  
    }  
  
    public void setMinuto(int minuto) {  
        this.minuto = minuto;  
    }  
  
    public String getOrario(){  
        return ora+":"+minuto;  
    }  
}
```

```
Orologio orologio = new Orologio( ora: 1200, minuto: -56);  
orologio.getOrario(); // 1200:-56
```

Posso scrivere un dato non coerente con la logica dell'oggetto

Utilizzo di una classe

```
public class Orologio {  
  
    public int ora;  
    public int minuto;  
  
    public String getOrario() {  
        return ora + ":" + minuto;  
    }  
  
}
```

```
public static void main(String[] args) {  
  
    Orologio orologio = new Orologio();  
    orologio.ora = 1200;  
    orologio.minuto = -56;  
    orologio.getOrario(); // 1200:-56  
  
}
```

Con i campi pubblici non ho modo di evitare che vengano scritti dati non coerenti con la logica dell'oggetto

Utilizzo di una classe

```
public class Orologio {  
  
    private int ora;  
    private int minuto;  
  
    public boolean setOra(int nuovaOra){  
        if (nuovaOra < 0 || nuovaOra > 23)  
            return false;  
  
        ora = nuovaOra;  
        return true;  
    }  
  
    public String getOrario(){  
        return ora+":"+minuto;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Orologio orologio = new Orologio();  
    orologio.setOra(20); //true  
    orologio.getOrario(); // 20:00  
  
    orologio.setOra(50); //false  
    orologio.getOrario(); // 20:00  
}
```

Con il setter di un campo privato ho modo di fare controlli di coerenza sul dato

Enumerazioni

Spesso può essere utile definire dei tipi di classe speciali, detti enumerazioni o enum, i cui valori possono essere scelti tra un insieme predefinito di identificatori univoci

I valori delle enum sono obbligatoriamente statici e sono associati a dei valori costanti

Un'enum non può essere istanziata, ma si può solo accedere ai suoi valori statici

Un'enum può, come le classi normali, avere costruttori, metodi e campi

```
public enum Enumerazione {  
    VALORE1, VALORE2, VALORE3  
}
```

```
public enum SemeCarta {  
    CUORI, QUADRI, FIORI, PICCHE  
}
```

Enumerazioni

Vogliamo rappresentare il concetto di mese, con un numero associato (1-12) e la rappresentazione come stringa (GEN, FEB, etc.)

```
public class Mese {  
    private int mese;  
  
    public Mese(int mese) {  
        this.mese = mese;  
    }  
  
    public int toInt(){  
        return mese;  
    }  
  
    public String toString(){  
        switch (mese){  
            case 1: return "GEN";  
            case 2: return "FEB";  
            /* ... */  
            case 12: return "DEC";  
            default: return null;  
        }  
    }  
}
```

Enumerazioni

Usando un'enumerazione

```
public enum Mese {  
    GEN(1), FEB(2), MAR(3), APR(4), MAG(5), GIU(6), LUG(7), AGO(8), SET(9), OTT(10), NOV(11), DIC(12);  
  
    private int mese;  
  
    Mese(int mese){  
        this.mese = mese;  
    }  
  
    public int toInt() {  
        return mese;  
    }  
}
```

Enumerazioni

Metodi inclusi

Enum.values()

Restituisce un array che
contiene ogni istanza dei valori
dell'enumerazione

Enum.valueOf(String value)

Restituisce l'istanza del valore
dell'enumerazione che
corrisponde alla stringa passata
in input

```
Mese[] mesi = Mese.values(); // GEN, FEB, MAR ...  
Mese m = Mese.valueOf("GEN"); // m = Mese.GEN
```

Enumerazioni

Integrazione diretta con il costrutto Switch

```
public static void main(String[] args) {  
  
    Mese m = null;  
  
    switch (m){  
        case GEN:  
            System.out.println("gennaio");  
            break;  
        case FEB:  
            System.out.println("febbraio");  
            break;  
        case MAR:  
            System.out.println("marzo");  
            break;  
        /*  
        ...  
        */  
    }  
}
```


Il Paradigma Object Oriented



Incapsulamento

- Il termine **incapsulamento** indica la proprietà che hanno gli oggetti di incorporare al loro interno sia le loro caratteristiche, i campi, sia le meccaniche del loro comportamento, i metodi. Tutto ciò che si riferisce ad un oggetto è contenuto in esso.
- Un oggetto espone all'esterno solo i metodi strettamente necessari per essere utilizzabile dall'esterno
- Non è quindi necessario conoscere il funzionamento di un oggetto per poterlo utilizzare, ma è necessario conoscere esclusivamente la sua **interfaccia**



Incapsulamento

- Data Hiding
 - Nascondere il dato dietro l'implementazione di un metodo
 - Uso di **Getter** e **Setter** per gli attributi
- Controllo sul dato
 - Il metodo setter non solo "setta" il dato, ma può fare un controllo di correttezza e consistenza del dato
- Dato calcolato
 - Un metodo getter non necessariamente ritorna il valore di un campo, potrebbe ritornare un dato calcolato o aumentato a partire da altri attributi

Il Paradigma Object Oriented

```
public class Orologio {

    private int ora;
    private int minuto;

    /**
     * @return true se l'ora è stata aggiornata, false se l'ora non è tra 0 e 23
     */
    public boolean setOra(int nuovaOra){
        if (nuovaOra < 0 || nuovaOra > 23)
            return false;

        ora = nuovaOra;
        return true;
    }

    /**
     * @return true se il minuto è stato settato, false se il minuto non è tra 0 e 59
     */
    public boolean setMinuto(int nuovoMinuto){
        if (nuovoMinuto < 0 || nuovoMinuto > 59)
            return false;

        minuto = nuovoMinuto;
        return true;
    }

    public String getOrario(){
        return ora + ":" + minuto;
    }

}
```

Il metodo setter effettua un controllo di coerenza del dato, impedendo un uso improprio della classe

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

    Orologio orologio = new Orologio();

    boolean oraSettata = orologio.setOra(19); // true
    boolean minutoSettato = orologio.setMinuto(20); // true
    orologio.getOrario(); // 19:20

    oraSettata = orologio.setOra(26); // false
    minutoSettato = orologio.setMinuto(130); // false
    orologio.getOrario(); // 19:20

}
```

Il Paradigma Object Oriented

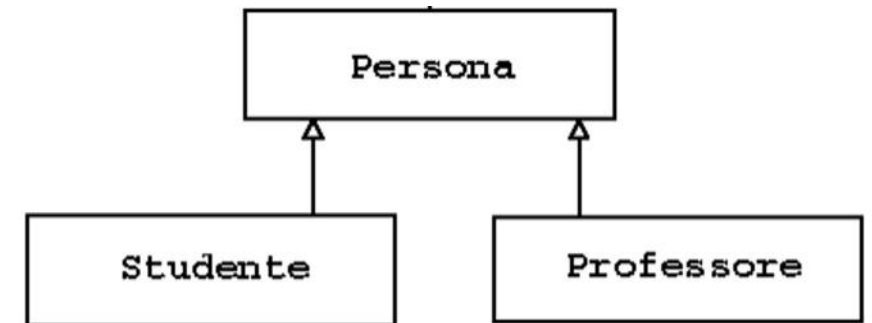
Una classe può avere un'implementazione contorta del suo funzionamento, ma questo non deve interessare l'utilizzatore

```
public class Orologio {  
  
    private int oraInMinuti;  
    private int minuto;  
  
    public boolean setOra(int nuovaOra){  
        if (nuovaOra < 0 || nuovaOra > 23)  
            return false;  
  
        oraInMinuti = nuovaOra*60;  
        return true;  
    }  
  
    public boolean setMinuto(int nuovoMinuto){  
        if (nuovoMinuto < 0 || nuovoMinuto > 59)  
            return false;  
  
        minuto = nuovoMinuto;  
        return true;  
    }  
}
```

```
    public int getOra(){  
        return oraInMinuti/60;  
    }  
  
    public int getMinuti(){  
        return minuto;  
    }  
  
    public String getOrario(){  
        return getOra() + ":" + getMinuti();  
    }  
  
    public String getOrario2(){  
        return (oraInMinuti/60) + ":" + minuto;  
    }  
}
```

Ereditarietà

- E' possibile creare un classe a partire da un'altra con un procedimento chiamato **estensione**, alla base del funzionamento dell'ereditarietà
- La nuova classe eredita campi e metodi della classe che estende
- Permette di riutilizzare gran parte del codice
- Studente e professore sono entrambi persone, condividono gran parte delle azioni (camminare, mangiare) e delle caratteristiche (nome, cognome)
- Ma differiscono per altre
 - Uno studente ha una matricola e può sostenere esami
 - Un professore dirige un corso e può esaminare uno studente



Ereditarietà

```
public class Persona {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
}
```

```
public class Professore extends Persona {  
  
    private String codiceCorso;  
  
    public String getCodiceCorso() {  
        return codiceCorso;  
    }  
}
```

```
public class Studente extends Persona {  
  
    private String matricola;  
  
    public String getMatricola() {  
        return matricola;  
    }  
}
```

```
public static void main(String[] args) {  
    Professore p = new Professore();  
    Studente s = new Studente();  
  
    p.getName();  
    p.getCodiceCorso();  
  
    s.getName();  
    s.getMatricola();  
}
```


Polimorfismo

L'obiettivo è poter utilizzare facilmente e senza differenziazioni formali oggetti diversi ma con caratteristiche comuni



- Nella OOP è possibile trattare allo stesso modo oggetti di classi diverse che estendono la stessa classe
- Se la classe Frutta espone il metodo mangia(), e le classi Pera e Pesca la estendono, io posso richiamare il metodo mangia() senza dover necessariamente sapere se l'oggetto è istanza di Pera o di Pesca, ma mi basta sapere che è un'istanza della classe Frutta

Polimorfismo

```
public class Frutto {  
  
}
```

```
public class Pera extends Frutto {  
  
}
```

```
public class Pesca extends Frutto {  
  
}
```

```
public static void main(String[] args) {  
    Pera p = new Pera();  
    Pesca p2 = new Pesca();  
    mangia(p);  
    mangia(p2);  
}  
  
public static void mangia(Frutto f){  
    // mangio il frutto  
}
```

Astrazione

- Una classe può essere astratta quando non ha senso istanziarla
- Deve essere estesa per poter essere usata
- Può esporre metodi astratti che vanno implementati dalle sottoclassi

```
public abstract class Frutto {  
  
    public abstract String getColore();  
  
}
```

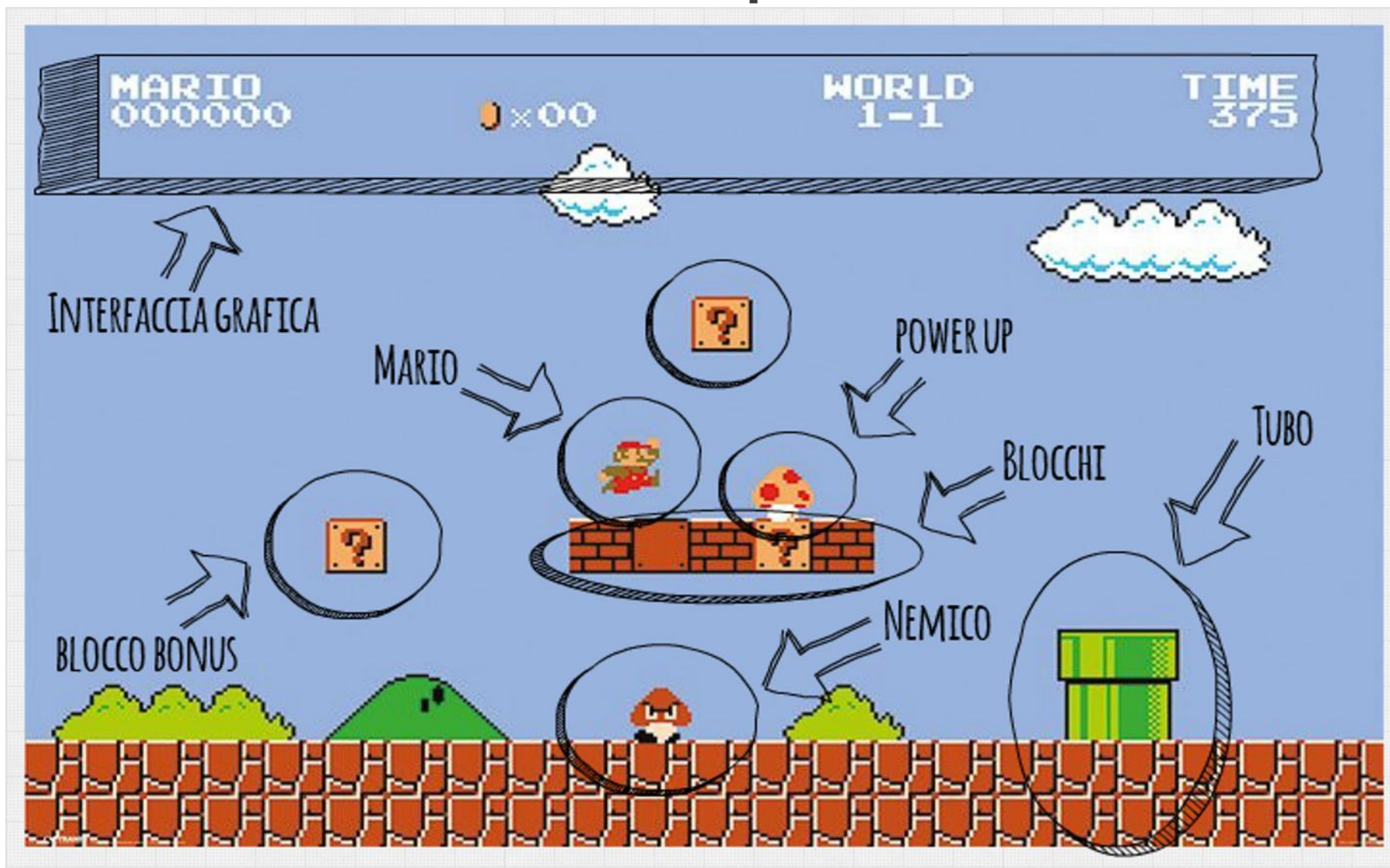
```
public class Pesca extends Frutto{  
  
    @Override  
    public String getColore() {  
        return "Rosa";  
    }  
}
```



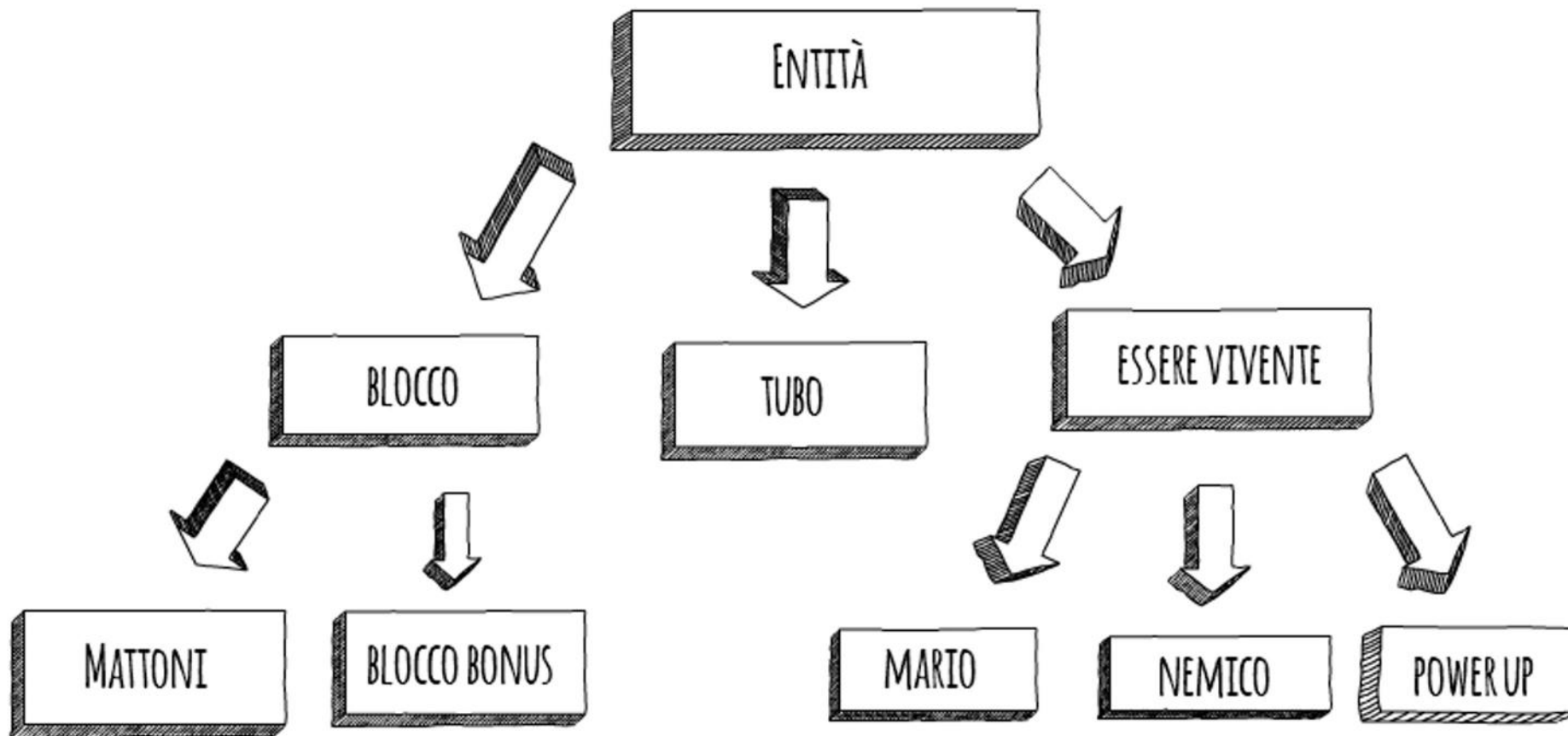
Esempio



Esempio



Esempio



Comparare oggetti

Come sappiamo, in Java l'operatore `==` compara le reference di due oggetti, mentre il metodo *`equals`* compara il contenuto effettivo (ossia lo stato composto dalle variabili)

Nella prossima lezione vedremo come ridefinire il metodo *`equals`* in modo appropriato

Ordinare Array di oggetti

La classe `Arrays` (fornita dalla JDK) mette a disposizione un metodo apposito per ordinare un array di oggetti:

```
Arrays.sort(arrayDiOggetti);
```

Nel caso di array formati da oggetti (e non primitivi), tali oggetti dovranno implementare l'interfaccia **Comparable** (torneremo su questo punto nel modulo 6!)

Il metodo `Arrays.sort` (come anche altri metodi), accetta in input, oltre all'array da ordinare, anche un oggetto opzionale di tipo `Comparator`, che può utilizzare per «capire» come ordinare gli elementi all'interno dell'array nel caso in cui questi non siano dei primitivi

```
Arrays.sort(arrayDiOggetti, comparator);
```

Esempio

```
int[] numbers = {4, 9, 1, 3, 2, 8, 7, 0, 6, 5};  
System.out.println(Arrays.toString(numbers));  
java.util.Arrays.sort(numbers);  
System.out.println(Arrays.toString(numbers));
```

Output:

Before sorting: [4, 9, 1, 3, 2, 8, 7, 0, 6, 5]

After sorting: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

(potete evitare di scrivere «java.util.», basta che accettiate l'import che vi suggerirà l'IDE!)

Domande?



Esercizi



Esercizio - Contatore

Progettare una classe Contatore che permetta di

- Istanziare la classe con un valore iniziale
- Istanziare la classe senza un valore iniziale
- Incrementare il conteggio attuale
- Ottenere il conteggio attuale
- Resetare il conteggio a zero
- Resetare il conteggio ad un altro valore

Scrivere un main che verifica le funzioni della classe Contatore

Esercizio - Forme

Progettare una classe Quadrato, che permetta di

- Istanziare la classe con la dimensione del lato del quadrato
- Ottenere il perimetro del quadrato
- Stampare il quadrato sulla console

Progettare una classe Cerchio, che permetta di

- Istanziare la classe con un costruttore che accetta un parametro
- Ottenere la circonferenza del cerchio
- Ottenere l'area del cerchio

Scrivere un main che verifica le funzioni delle classi

Esercizio - Lampadina

Progettare una classe Lampadina che rappresenti una lampadina elettrica

- La lampadina può essere accesa, spenta o rotta
- Espone due metodi
 - stato() che indica lo stato corrente della lampadina
 - click() che cambia lo stato da accesa a spenta o da spenta ad accesa, oppure rompe la lampadina
 - Una lampadina si rompe dopo un numero di click definito dal produttore
- La classe deve contenere uno o più campi che ne descrivano lo stato
- Un costruttore
- I metodi indicati sopra

Scrivere un main che verifica le funzioni delle classi

Esercizio – Lampadina2

Progettare una classe Interruttore che rappresenta un interruttore per la lampadina fatta precedentemente

- Ogni interruttore è collegato ad una lampadina e ne regola accensione e spegnimento
- Definire quali campi, metodi e costruttori siano opportuni
- Creare un metodo di test che istanzia due interruttori e li collega alla stessa lampadina e poi offre all'utente ripetutamente la possibilità di clickare uno dei due interruttori oppure di terminare l'esecuzione

Esercizio – Lampadina3

Modificare la classe Lampadina facendo in modo che tutte le lampadine condividano l'informazione sulla presenza di corrente all'interno dell'impianto (immaginate che tutte le lampadine siano collegate allo stesso impianto di corrente)

Le lampadine devono comportarsi coerentemente con la presenza o meno di elettricità nell'impianto

Quindi quando non c'è corrente una lampadina può essere soltanto nello stato «spento» o «rotto»

Scrivere un metodo di test che testi la funzione di "staccare" o "riattaccare" la corrente