tree **school**

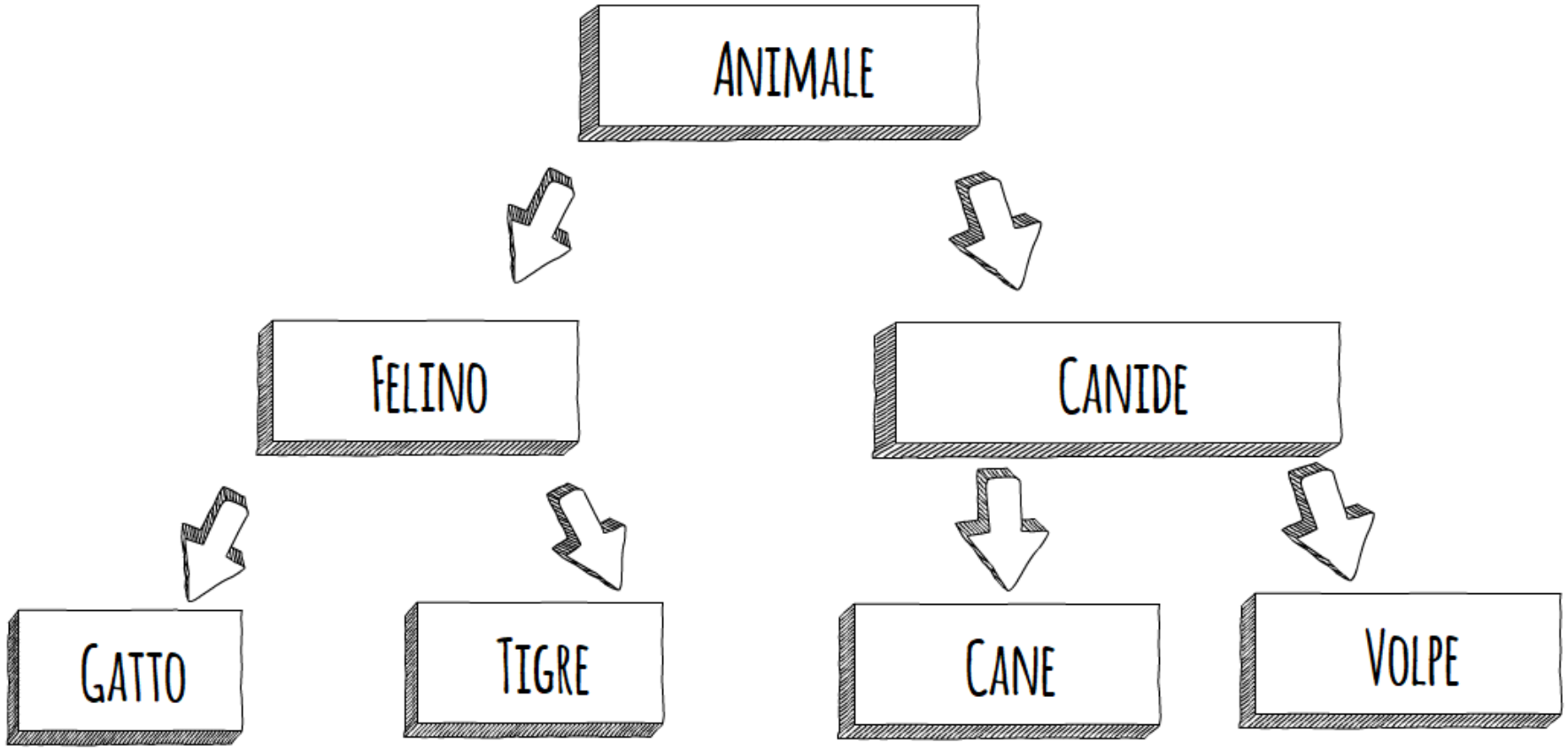
Corso Java Backend
Modulo 6
Interfacce, astrazione e classi nidificate

Andrea Gasparini

Di cosa parliamo

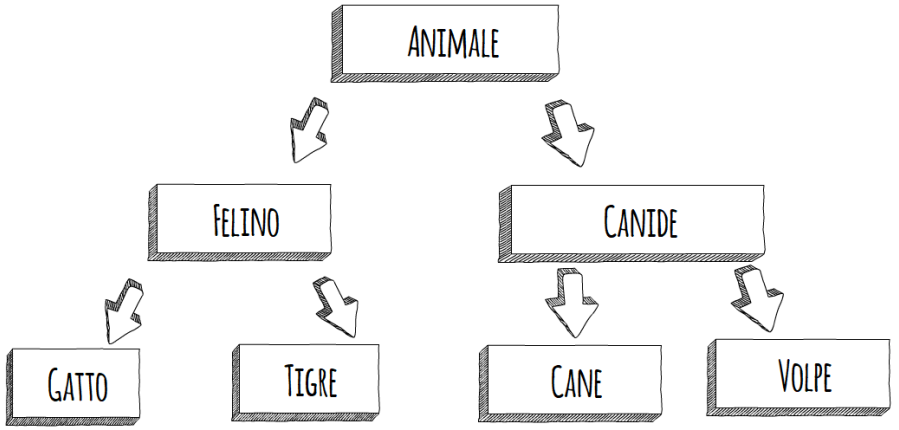
Di cosa parliamo

Un classico esempio



Dubbi legittimi

- **Cosa succedesse se instanziassi un animale?**
- **Cosa invece se instanziassi un felino?**
- **Dovrei volerle queste possibilità o dovrei vietare agli utenti di farlo?**

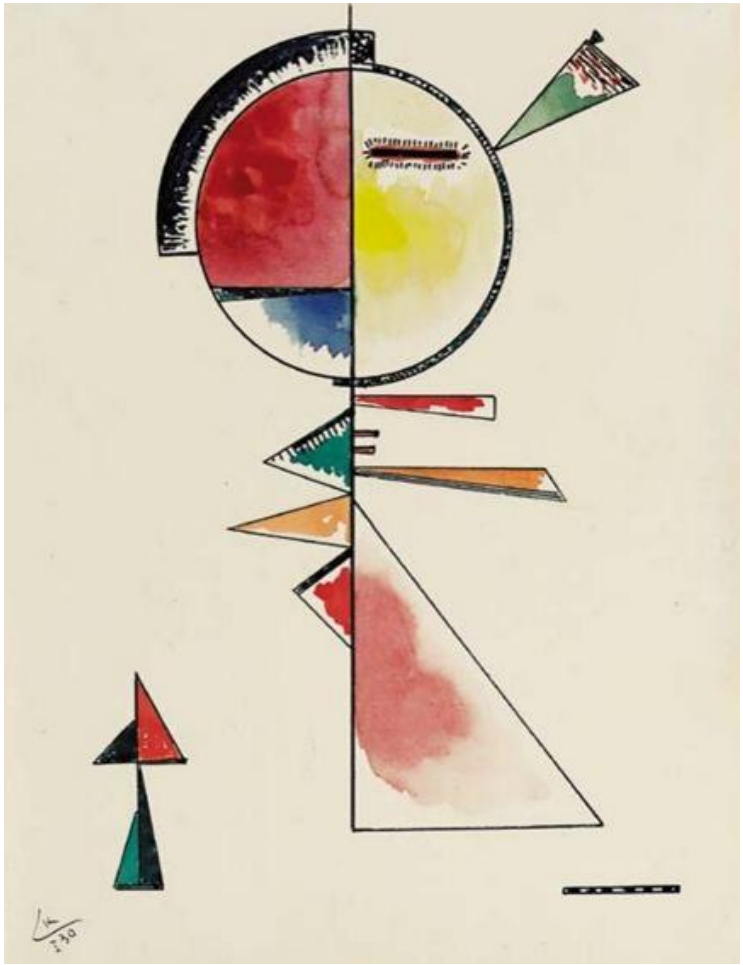


Classi Astratte

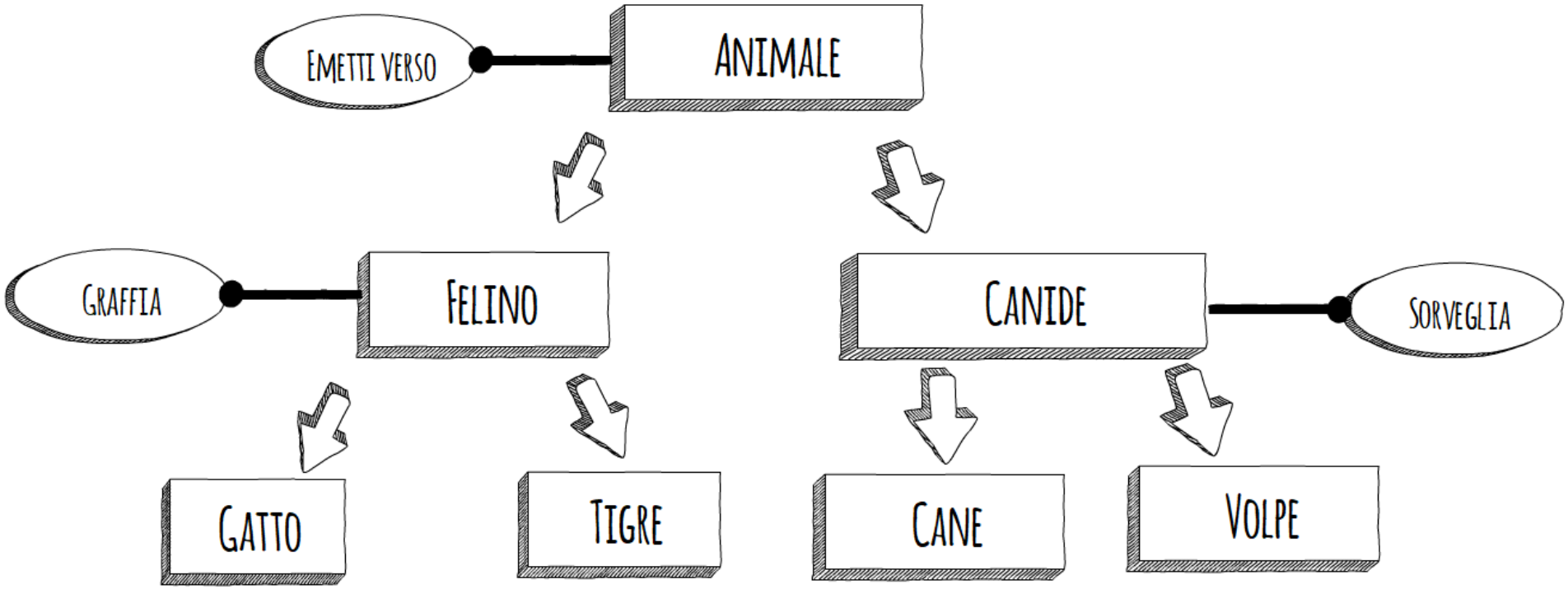
- Quando modelliamo un problema potremmo non volere che gli utenti stanzino classi «di mezzo».
- Le ragioni possono essere molteplici e sensate, pensate a una classe che modella una «generica» connessione ad un database.
- Questa classe potrebbe esporre dei metodi generici in comune con ogni database (apri connessione, inserisci dato, leggi dato, cancella etc..)
- I vari tipi di database presenti sul mercato hanno sempre qualche comando particolare o qualche feature che si differenzia dagli altri.
- Ha molto senso quindi che per ogni tipo di Database esista una classe specializzata che estenda la nostra classe base.
- Che comportamento vi aspettereste se istanziassimo la classe base?

Classi Astratte

- La OOP ci offre la possibilità di definire una classe «astratta».
- Una classe astratta è una classe che non può essere istanziata ma può essere estesa.
- Utilizziamo le classi astratte quando vogliamo creare una classe che modelli dei comportamenti generici ma che non ha senso di esistere senza una vera e propria specializzazione
- Grazie al polimorfismo possiamo comunque utilizzare un oggetto trattandolo come se fosse del tipo della classe astratta che estende.

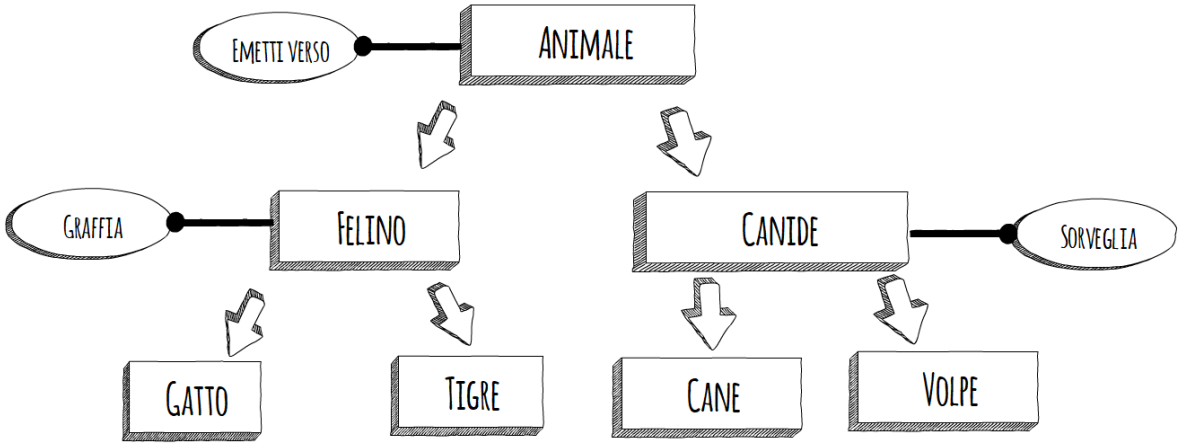


E ora?


EMETTI VERSO
ANIMALE
GRAFFIA
FELINO
CANIDE
SORVEGLIA
GATTO
TIGRE
CANE
VOLPE

Altri dubbi

- **Tutti gli animali che sto modellando hanno un verso ma ognuno ce l'ha diverso**
- **Gatti e tigri graffiano, ma il graffio delle tigri fa più male**
- **Cani e volpi sorvegliano in modo diverso**



Metodi astratti

- **Le classi astratte possono avere dei metodi astratti.**
- **I metodi astratti in quanto tali non hanno definizione ed è compito delle sottoclassi dargliene una.**
- **I metodi astratti ci permettono di poter modellare quei comportamenti che tutte le sottoclassi hanno ma che ognuna poi fa a modo suo**
- **Tutti i database permettono di inserire i dati, ma ogni database potrebbe avere il proprio modo personale per farlo.**

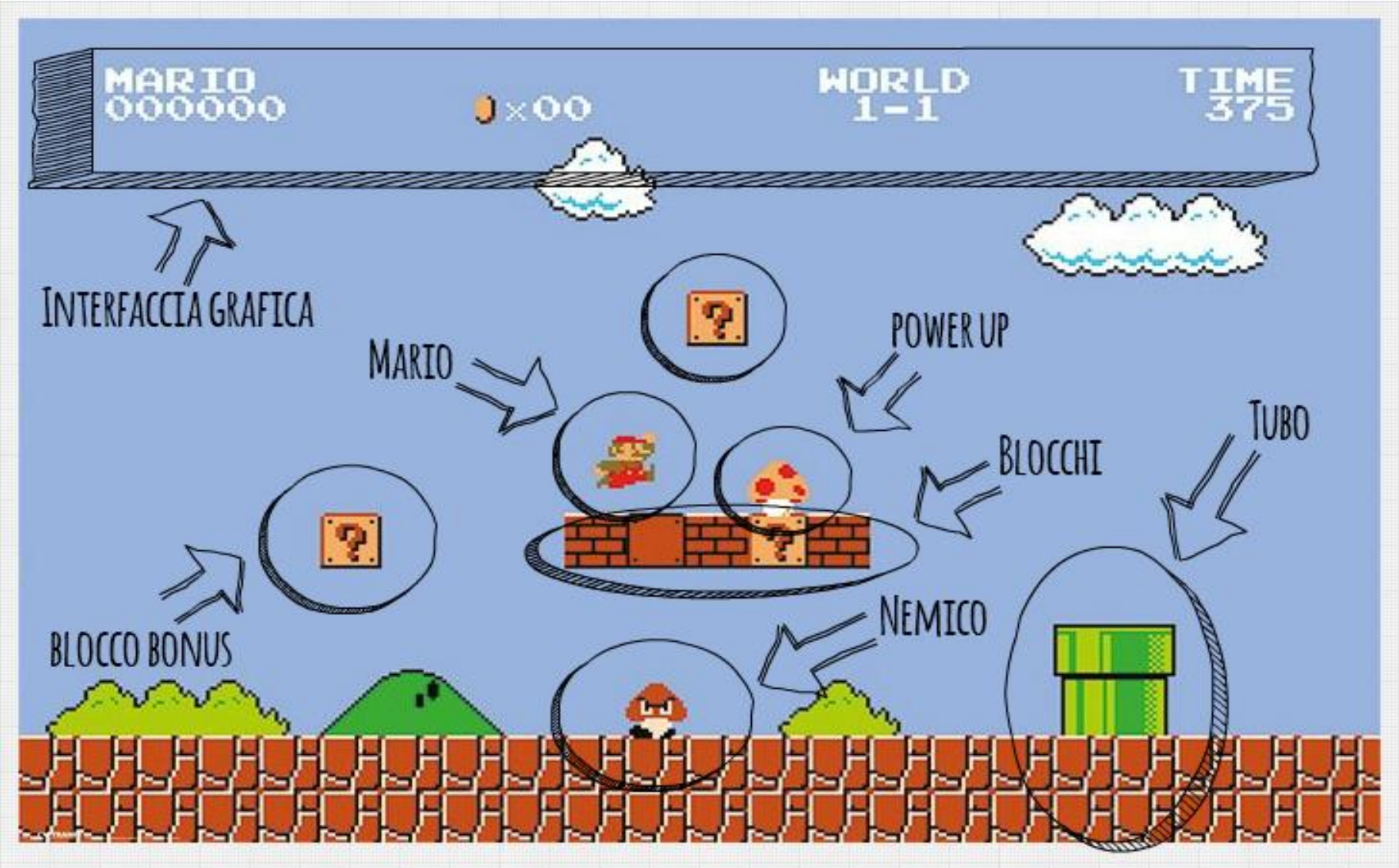
Classi Astratte


```java
public abstract class Frutto {

    private String nome;

    protected Frutto(String nome){
        this.nome = nome;
    }

    public String getNome(){
        return nome;
    }

    public void setNome(String nome){
        this.nome = nome;
    }

    public abstract void mangia();

}
```

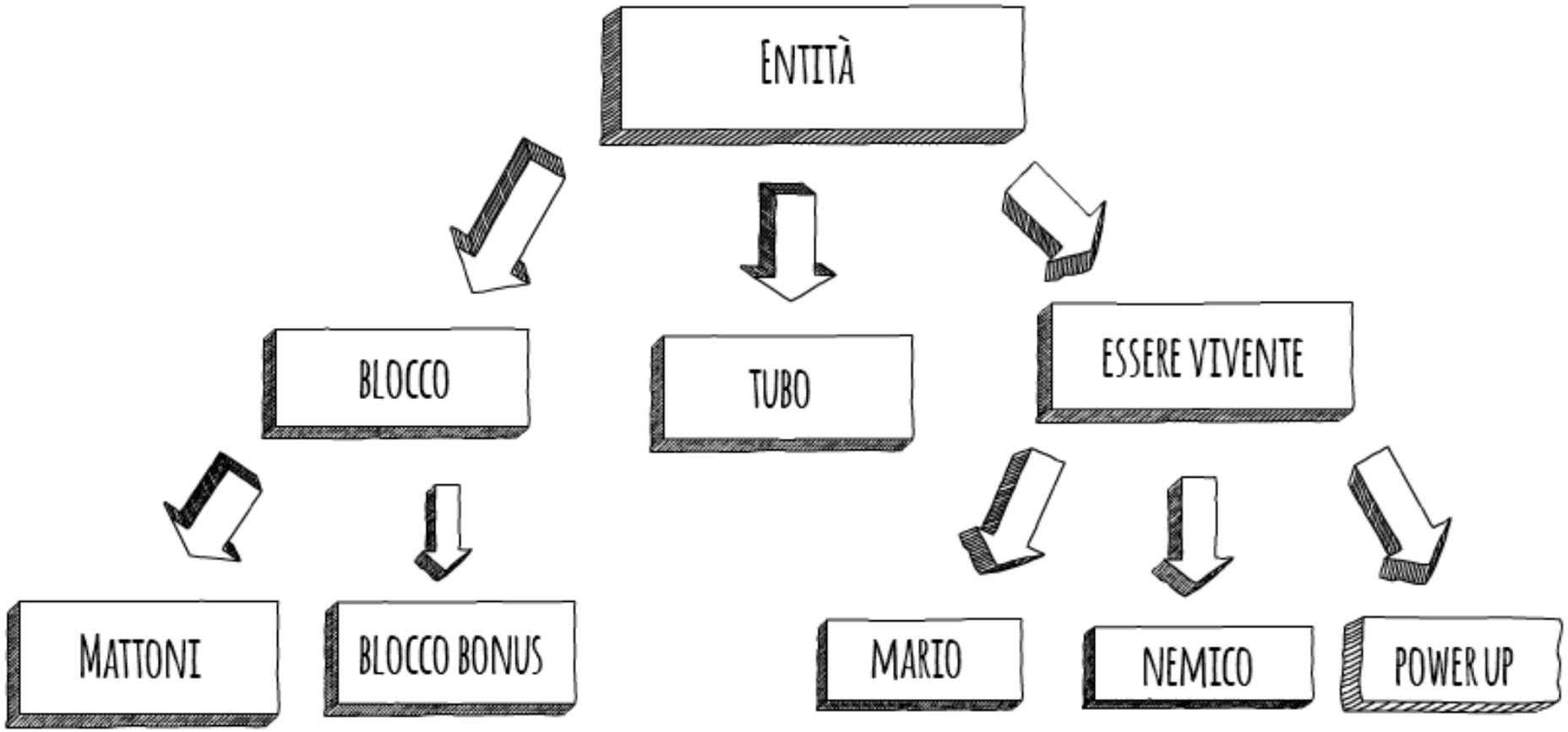
```java
public class Mela extends Frutto {

    public Mela() {
        super( nome: "Mela");
    }

    @Override
    public void mangia() {
        //mangia a morsi
    }

}
```

```java
public class Mandarino extends Frutto {

    public Mandarino() {
        super( nome: "Mandarino");
    }

    @Override
    public void mangia() {
        //mangia uno spicchio alla volta
    }

}
```

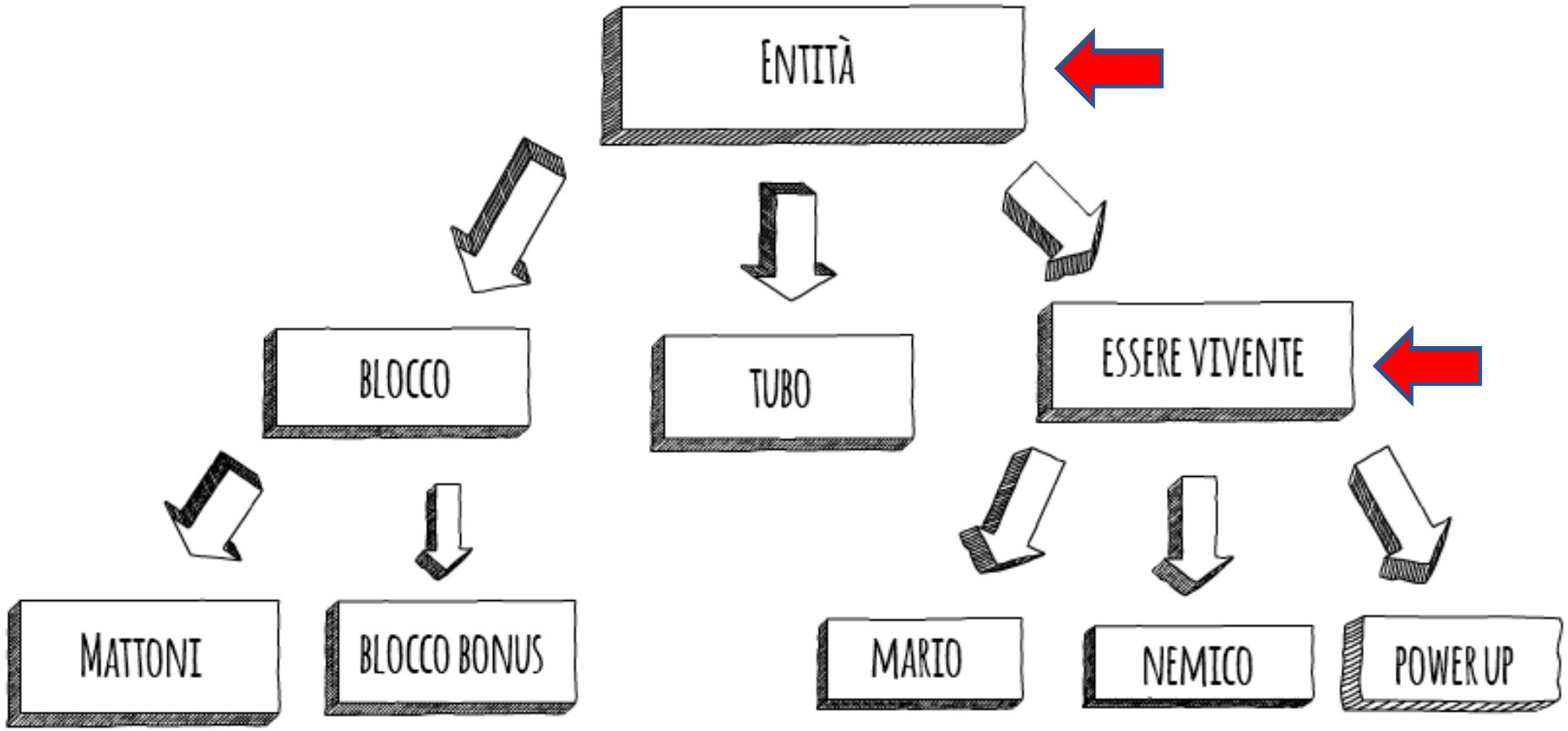
Esempio


MARIO
000000
×00
WORLD
1-1
TIME
375
INTERFACCIA GRAFICA
MARIO
POWER UP
BLOCCHI
TUBO
BLOCCO BONUS
NEMICO

Quali potrebbero essere astratte?



Una possibile risposta



Di cosa parliamo

Interfacce

- **Un’interfaccia è un mezzo attraverso il quale utilizzare un oggetto, quindi l’astrazione di un comportamento**

- **Se so usare un telecomando, posso utilizzare un televisore senza sapere come funziona all’interno.**

Interfacce

- **Le interfacce sono strumenti che permettono a più classi di fornire ed implementare un insieme di metodi che hanno in comune**

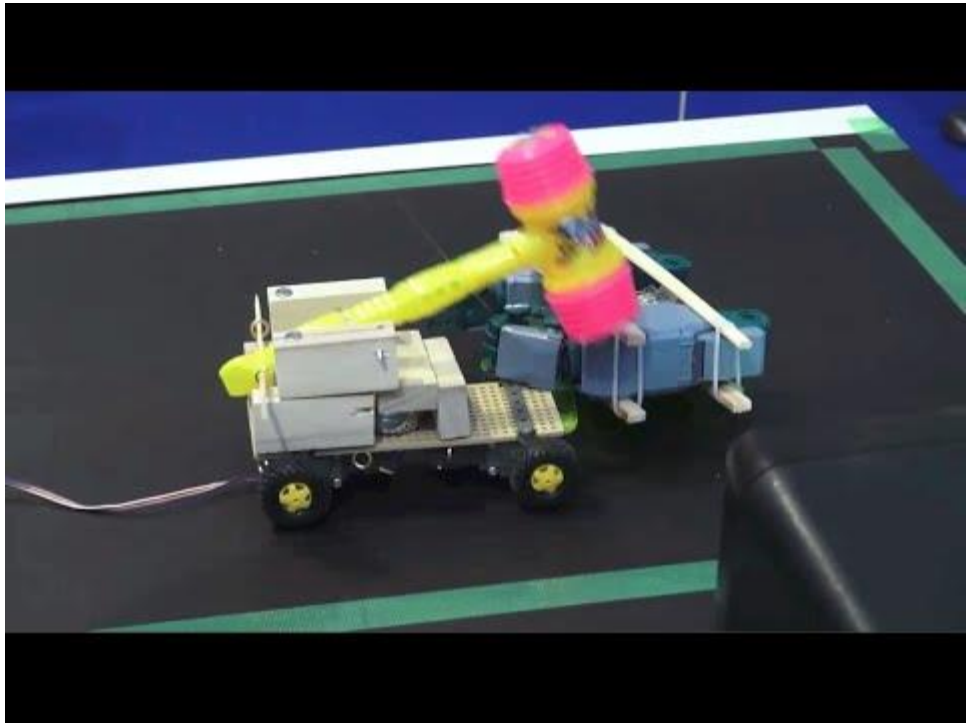
- **Un'interfaccia specifica un comportamento ma non lo implementa, perché ogni oggetto potrebbe avere un modo differente di eseguirlo**

A cosa potrebbero servirmi

- **Stiamo progettando un videogioco di battaglie tra mecha e abbiamo la necessità di ordinarli per grandezza**
- **Conosciamo diversi algoritmi di ordinamento, alcuni li abbiamo scritti noi e Java ce ne offre qualcuno già implementato… Come faccio però ad usarli con la classe Mecha senza riscrivere il codice dell'algoritmo di ordinamento?**
- **Java ci offre un'interfaccia chiamata Comparable che forza le classi che la implementano a implementare un metodo CompareTo che prende in input un oggetto dello stesso tipo e ritorna se è più piccolo, più grande o uguale.**
- **Un algoritmo di ordinamento che ordina oggetti di tipo Comparable è quindi in grado di ordinare qualsiasi tipo di oggetto a patto che questi implementi l'interfaccia Comparable**

Interfacce VS Classi astratte

- **La domanda che nasce spontanea ora è: Come faccio a capire quando è meglio utilizzare una classe astratta o un'interfaccia?**
- **Nel caso dell'ordinamento, se usassimo una classe astratta tutte le classi che vogliamo poter ordinare dovrebbero in un modo o nell'altro ereditare la classe base e questo non è proprio comodissimo**
- **Nel caso invece dell'esempio dei database, la classe base potrebbe contenere all'interno della logica in comune riguardo i database e quindi potremmo preferire di utilizzare una classe astratta.**



Interfacce

Un'interfaccia è tipo astratto di dato che può contenere:

- **Attributi**
- **Attributi costanti**
- **Metodi astratti**
- **Metodi default (java 8+)**
- **Metodi statici (java 8+)**
- **Metodi privati (java 9+)**

```java
public interface Interfaccia {

    public static final int value = 5;
    int value2 = 10;

    public abstract void metodoAstratto();

    public default void metodoDefault(){
        //do something
    }

    public static void metodoStatico(){
        //do domething
    }

    private void metodoPrivato(){
        //do something
    }


}
```

Interfacce: cosa posso o non posso fare?

- **Per utilizzarle devo usare la keyword implements e in seguito scrivere il nome dell'interfaccia**
- **Devo obbligatoriamente implementare i metodi astratti**
- **Posso ridefinire i metodi default, se necessario**
- **Posso accedere alle costanti**
- **Non posso accedere o ridefinire i metodi privati**
- **Come le classi astratte, non posso istanziare direttamente un'interfaccia**

```java
public class MiaClasse implements Interfaccia {

    @Override
    public void metodoAstratto() {
        //implementa metodo astratto
    }

    public void faiQualcosa(){
        metodoAstratto();
        metodoDefault();
        int x = value;
        metodoPrivato();
    }


}
```

Interfacce esempio


```
public interface SupportoRiscrivibile {

    void leggi();
    void scrivi();

}
```

- **L'interfaccia SupportoRiscrivibile definisce il comportamento di un qualcosa che può essere letto e scritto esponendo i metodi leggi() e scrivi()**

Interfacce esempio


```java
public class DVD implements SupportoRiscrivibile {

    @Override
    public void leggi() {
        avviaMotore();
        spostaLaser();
        leggiCella();
    }

    @Override
    public void scrivi() {
        avviaMotore();
        spostaLaser();
        scriviCella();
    }

    private void avviaMotore(){}

    private void spostaLaser(){}

    private void leggiCella(){}

    private void scriviCella(){}
}
```

- **La classe DVD implementa il comportamento descritto dall'interfaccia SupportoRiscrivibile**
- **Definisce il comportamento dei metodi leggi e scrivi in base al proprio funzionamento**

Interfacce esempio


```java
public class PenDriveUSB implements SupportoRiscrivibile{

    @Override
    public void leggi() {
        accediAllaDirectory();
    }

    @Override
    public void scrivi() {
        scriviNelPercorso();
    }

    private void accediAllaDirectory(){}
    private void scriviNelPercorso(){}
}
```

- **La classe PenDriveUSB implementa il comportamento descritto dall'interfaccia SupportoRiscrivibile**
- **Definisce il comportamento dei metodi leggi e scrivi in base al proprio funzionamento, che è differente da quello della classe DVD**

Interfacce esempio


```java
public class Quaderno implements SupportoRiscrivibile {

    @Override
    public void leggi() {
        //leggi dal quaderno
    }

    @Override
    public void scrivi() {
        //scrivi con la penna sul quaderno
    }
}
```

- **Anche la classe Quaderno è un supporto riscrivibile**
- **Le interfacce permettono di modellare comportamenti comuni a classi che non appartengono allo stesso ordine gerarchico (is-a)**

Interfacce, multi implementazione

- **Con Java non è consentito per una classe estendere più classi, ma l'estensione è limitata ad una singola superclasse**
- **Al contrario è possibile invece implementare un qualsiasi numero di interfacce**
- **In aggiunta è possibile contemporaneamente estendere 1 classe ed implementare n interfacce**

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
```

Interfacce, ereditarietà

- **È possibile creare un interfaccia partendo da un'altra interfaccia utilizzando l'ereditarietà**
- **L'interfaccia erediterà tutte le astrazioni contenute nell'interfaccia padre**
- **Un'interfaccia può ereditare da più interfacce.**

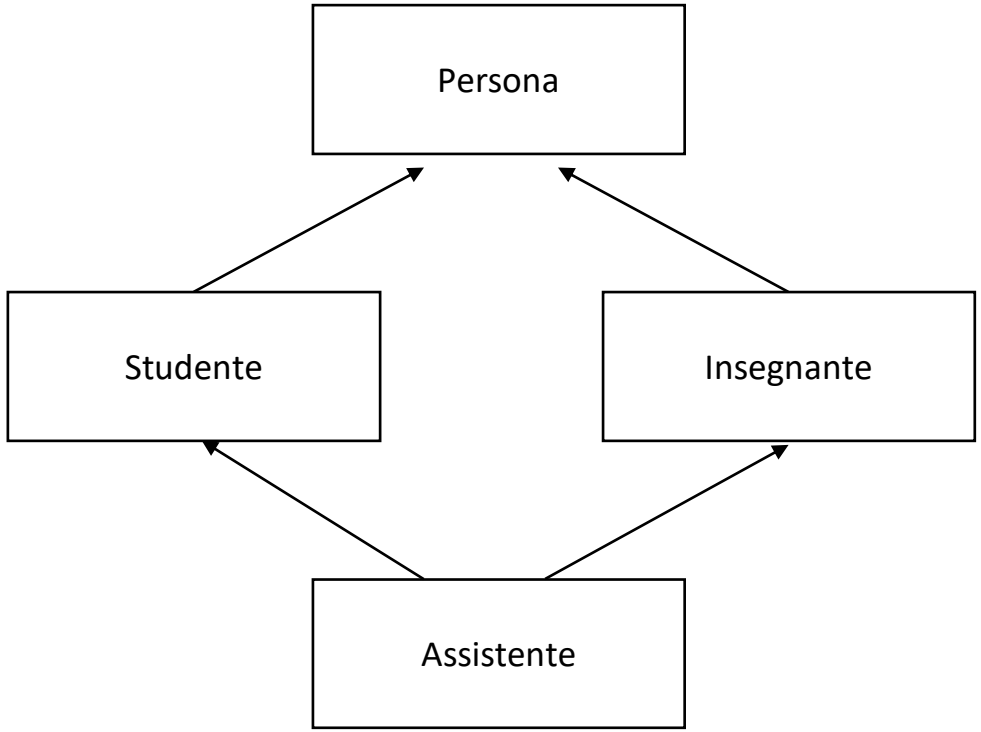
```
public interface ASD extends Comparable, Serializable {
```

Interfacce

Perché posso implementare quante interfacce voglio ma estendere una sola classe?

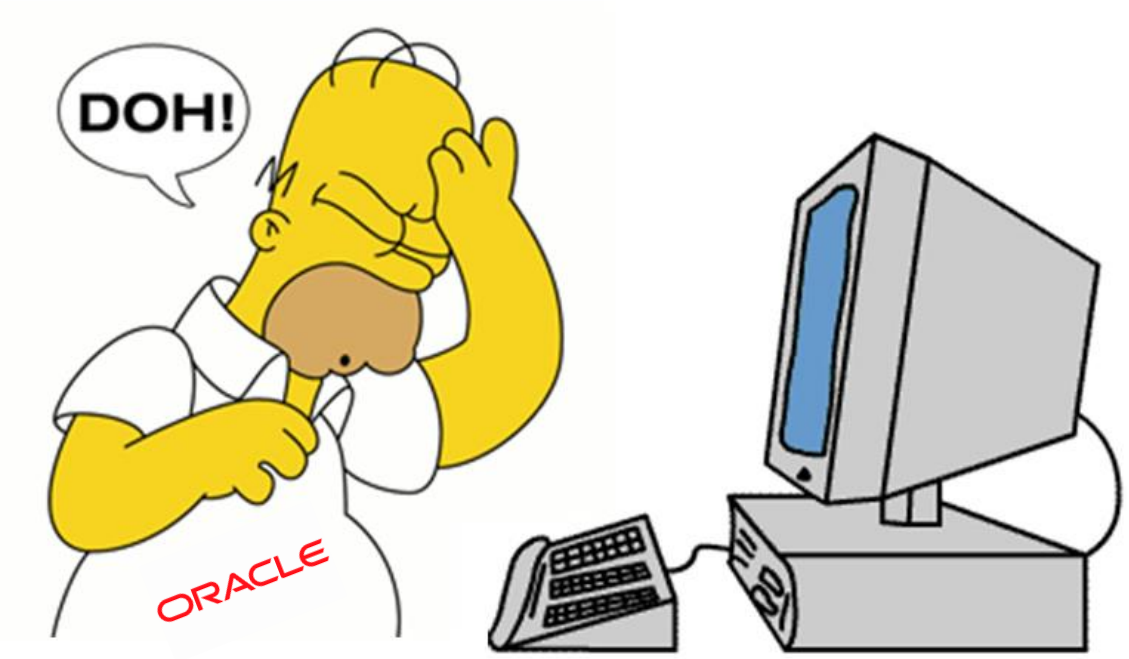
The diamond problem

- **Il problema del diamante riguarda sia la logica della OOP che anche alcuni comportamenti interni del compilatore**
- **In questo caso specifico, se studente e insegnante hanno un metodo che si chiama «leggi libro», quale versione eredita Assistente?**
- **Se Persona ha un attributo altezza e sia studente che insegnante lo ereditano, assistente avrà due altezze.**



The diamond problem

- **Fino a Java 8 le interfacce potevano avere solo metodi astratti e il problema del «quale eredito» era risolto perché tanto qualsiasi metodo ereditavo comunque avrei dovuto riscriverlo.**
- **Con Java 8 l'introduzione dei metodi di default delle interfacce ha fatto tornare l'annosa questione del diamond problem.**
- **Il problema viene risolto con particolari regole sulla precedenza, in alcuni casi però queste regole non riescono a risolvere il problema e in questi casi il compilatore non compila e ci da un errore**
- **Questo modo di gestire il problema ricorda molto quello che utilizza il cpp…**



The diamond problem

- **Come facciamo se la mia classe implementa due interfacce che hanno metodi con lo stesso nome?**

```java
public interface Interfaccia1 {

    void metodoAstratto();

    default void metodoDefault(){
        System.out.println("interfaccia1");
    }

}
```

```java
public interface Interfaccia2 {

    void metodoAstratto();

    default void metodoDefault(){
        System.out.println("interfaccia2");
    }

}
```

```java
public class MiaClasse implements Interfaccia1, Interfaccia2 {

    @Override
    public void metodoAstratto() {
        //implementa metodo astratto
    }

    @Override
    public void metodoDefault() {
        Interfaccia1.super.metodoDefault();
        Interfaccia2.super.metodoDefault();
    }


}
```

Polimorfismo nelle interfacce

- **Anche per le interfacce si può sfruttare il polimorfismo, quindi un oggetto di una classe che implementa un'interfaccia, può essere utilizzato come se fosse un'istanza di quell'interfaccia**

```
public class MiaClasse implements Interfaccia {

    @Override
    public void metodoAstratto() {
        //implementa metodo astratto
    }

    public void metodoDellaClasse(){

    }

}
```

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

    Interfaccia x = new MiaClasse();
    x.metodoAstratto();
    x.metodoDellaClasse();
```

Di cosa parliamo

Classi interne e nidificate

Le classi che abbiamo visto fino ad ora sono tutte classi definite top-level, perché si trovano ad un livello più alto rispetto a tutte le altre classi

Si trovano tutte all'interno del proprio file .java e non sono contenute in altre classi

Però è possibile creare classi interne ad altre classi, prendendo il nome di classi annidate (nested class)

Le classi nidificate possono essere static o non-static, in quest'ultimo caso vengono chiamate classi interne (inner class)

Classi interne

Classi Nidificate

Per poter definire una classe interna è necessario prima definire una classe top-level nella quale definire la nostra classe interna

Definire una classe internamente ad un'altra crea un legame implicito tra le due classi, ed infatti la classe interna ha sempre un riferimento implicito alla classe esterna, ed ha quindi la possibilità in ogni momento di accedere a metodi e campi della classe esterna

Una classe interna è a tutti gli effetti un membro della classe esterna, e quindi può essere dichiarato public, private o protected

Classi interne

Classi Nidificate

L'accesso a campi e metodi, sia dell'oggetto della classe interna che dell'oggetto della classe esterna a cui è legato a doppio filo, avviene normalmente

Nei casi di ambiguità (es. campi con lo stesso nome nella classe interna ed esterna):

- **L’uso del this all’interno della classe interna è un riferimento alla classe interna stessa**
- **Per ottenere il riferimento alla classe esterna è necessario preporre il nome della classe esterna: ClasseEsterna.this**

Classi interne

Classi Nidificate

```java
public class Tastiera {

    private String tipo;
    private Tasto[] tasti;

    public class Tasto{

        private char c;

        public Tasto(char c) {
            this.c = c;
        }

        public char premi(){
            return c;
        }

        public String getTipoTastiera(){
            return Tastiera.this.tipo;
        }

    }

    public Tastiera(String tipo, char[] caratteri){
        this.tipo = tipo;
        tasti = new Tasto[caratteri.length];

        for (int i = 0; i < caratteri.length; i++) {
            tasti[i] = new Tasto(caratteri[i]);
        }

    }

}
```

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

    Tastiera t = new Tastiera( tipo: "qwerty", new char[10]);
    Tastiera.Tasto t2 = t.new Tasto( c: 'c');

}
```

Per istanziare la classe interna da un'altra classe, è necessario un riferimento ad un oggetto della classe esterna

Tasto ha accesso al campo (privato!) della classe esterna

Posso anche accedere al campo direttamente tramite il nome (tipo, invece di Tastiera.this.tipo)

Classi nidificate (statiche)

Se una classe interna è *statica*, allora è definita Classe Nidificata (nested class)

In quanto "static" segue le regole dei campi e dei metodi statici, ovvero esiste senza la necessità che sia stata istanziata la classe esterna e non può accedere a tutto ciò che non sia dichiarato statico all'interno della classe esterna

Classi nidificate (statiche)


```java
public class Tastiera {

    private String tipo;
    private static String test;
    private Tasto[] tasti;

    public static class Tasto{

        private char c;

        public Tasto(char c) {
            this.c = c;
        }

        public char premi(){
            return c;
        }

        public String getTipoTastiera(){
            String x = Tastiera.test;
            return Tastiera.this.tipo;
        }

    }

    public Tastiera(String tipo, char[] caratteri){
        this.tipo = tipo;
        tasti = new Tasto[caratteri.length];

        for (int i = 0; i < caratteri.length; i++) {
            tasti[i] = new Tasto(caratteri[i]);
        }

    }

}
```

```java
public static void main(String[] args) {
    Tastiera.Tasto t = new Tastiera.Tasto( c: 'c');
}
```

Errore in fase di compilazione: la classe static Tasto non ha accesso implicito ai membri della classe esterna

Classi interne e nidificate perché?

Classi Nidificate

Per raggruppare le classi secondo un criterio logico, nel caso in cui una classe sia utile soltanto ad un'altra classe (es. Tasto per Tastiera), ha più senso raggrupparle insieme

Migliora la leggibilità del codice, perché due classi che lavorano insieme hanno il codice scritto in posti vicini (vicinanza spaziale)

Incrementa l'incapsulamento, dato che posso usare classi aggiuntive per implementare il comportamento della classe esterna

Classi anonime

Classi anonime

In Java è possibile definire classi anonime che implementano un'interfaccia o estendono una classe

Si usano per creare una singola istanza al volo secondo la necessità del codice

```java
@FunctionalInterface
public interface Runnable {
    /** When an object implement
    public abstract void run();
}
```

```java
public class PrintaCiao implements Runnable{

    @Override
    public void run() {
        System.out.println("ciao");
    }
}
```

```java
public static void main(String[] args) {
    Runnable r = new Runnable();
}
```

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

    Runnable r = new PrintaCiao();
    r.run();
}
```

Classi anonime

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

    Runnable r = new Runnable(){

        @Override
        public void run() {
            System.out.println("ciao");
        }
    };

    r.run();
}
```

ArrayList

ArrayList

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

    Integer[] array = new Integer[10];
    ArrayList<Integer> lista = new ArrayList<>();
    lista.size(); // 0
    lista.add(5);
    lista.size(); // 1
    lista.add(10);
    lista.toString(); // [5, 10]

    lista.get(0); // 5

    Integer x = 30;
    lista.add(x); // true

    lista.contains(x); // true

    lista.clear();
    lista.indexOf(x); // 2

    lista.remove( index: 0); // 5
    lista.remove(x); // true

    lista.set(10, 89);

    ArrayList<Integer> lista2 = new ArrayList<>();
    lista.addAll(lista2);
    lista.removeAll(lista2);


    lista.subList(2, 9);


}
```

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

    ArrayList<Integer> lista = new ArrayList<>();

    for (int i = 0; i < lista.size(); i++) {
        System.out.println(lista.get(i));
    }

    for (Integer elemento : lista) {
        System.out.println(elemento);
    }

}
```