



# ereditarietà e polimorfismo

Java

- l'ereditarietà permette di **definire nuove classi** partendo da classi sviluppate in precedenza
- la nuova classe viene definita esprimendo solamente le **differenze** che essa possiede rispetto alla classe di partenza
- l'ereditarietà permette di specificare “il punto di partenza”, cioè la **classe base**, e le **differenze** rispetto a questa

## Libro

```
-autore : string
-titolo : string
-numeroPagine : int
-codiceISBN : string

+setautore(in a : string)
+getautore() : string
+settitolo(in t : string)
+gettitolo() : string
+setnumeroPagine(in np : int)
+getnumeroPagine() : int
+setcodiceISBN(in c : string)
+getcodiceISBN() : string
+visualizza()
```

## LibroDiTesto

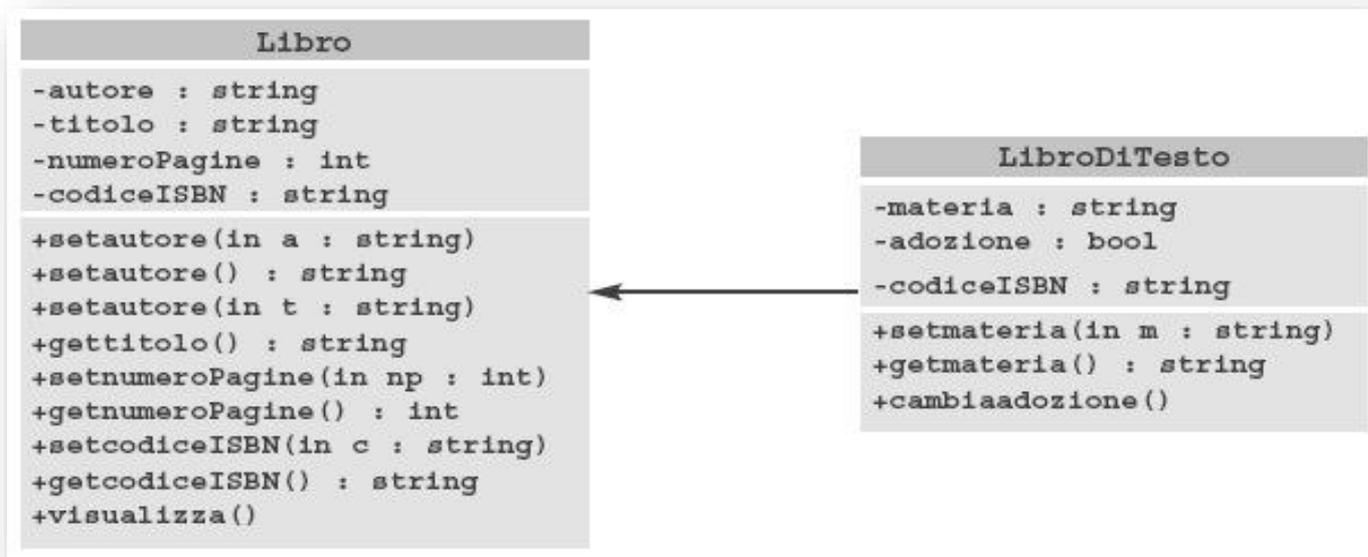
```
-autore : string
-titolo : string
-numeroPagine : int
-codiceISBN : string
-materia : string
-adozione : bool

+setautore(in a : string)
+getautore() : string
+settitolo(in t : string)
+gettitolo() : string
+setnumeroPagine(in np : int)
+getnumeroPagine() : int
+setcodiceISBN(in c : string)
+getcodiceISBN() : string
+visualizza()
+setmateria(in m : string)
+getmateria() : string
+cambiaadozione()
```

# un diagramma di ereditarietà

A. Ferrari

- LibroDiTesto *deriva* da Libro e *aggiunge* nuove caratteristiche



- la nuova classe (nell'esempio LibroDiTesto) viene definita **sottoclasse** (o **classe derivata**)
- la classe di provenienza (nell'esempio Libro) viene definita **superclasse** (o **classe base**)
- la sottoclasse *eredita* tutte le caratteristiche (attributi e metodi) della superclasse e si differenzia da questa:
  - per l'*aggiunta* di nuovi attributi e/o metodi
  - per la *ridefinizione* di alcuni metodi della superclasse

```
class Sottoclasse extends Superclasse {  
    <attributi>  
    <metodi>  
}
```

```
class LibroDiTesto extends Libro {  
  
    private String materia;  
    private boolean adozione;  
  
    public void setmateria(String materia) {  
        this.materia = materia;  
    }  
  
    public String getmateria() {  
        return materia;  
    }  
  
    public void cambiaadozione() {  
        adozione = !adozione;  
    }  
}
```

# ereditarietà e modificatori di visibilità

A. Ferrari

- ***public***
  - elemento visibile in qualunque classe
- ***private***
  - elemento visibile solo all'interno della classe
- ***protected***
  - elemento visibile all'interno della classe e di tutte le sottoclassi di questa

# istanze di classe e di sottoclasse

A. Ferrari

- un **oggetto** di tipo **Sottoclasse** è contemporaneamente e automaticamente **anche** di tipo **Superclasse**
  - quando è necessario utilizzare un oggetto di tipo Superclasse è possibile utilizzare un oggetto di tipo Sottoclasse
- al **contrario** invece la regola **non vale**
- ogni oggetto di tipo LibroDiTesto è anche un oggetto di tipo Libro
  - *è vero che un libro di testo è un libro*
- non è vero il contrario
  - *un libro non è necessariamente un libro di testo, potrebbe essere per esempio un romanzo*

```
Libro lib;  
LibroDiTesto libtes = new LibroDiTesto();  
lib = libtes;
```

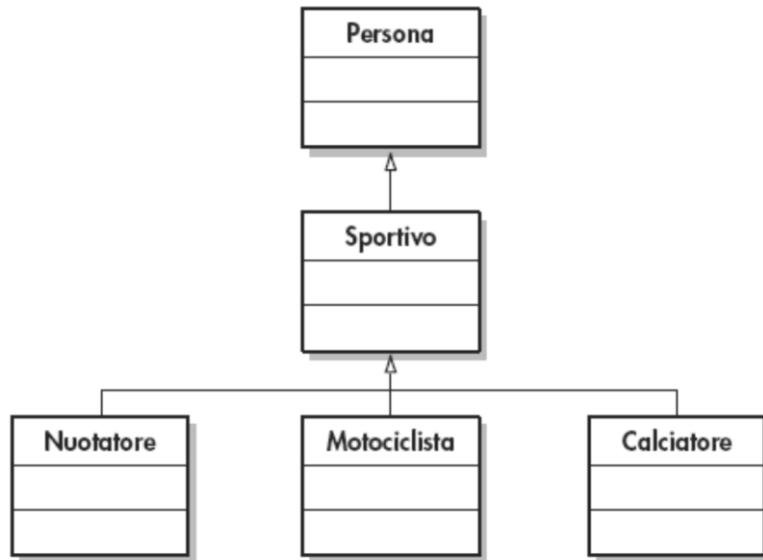
- il tipo `LibroDiTesto` è compatibile con il tipo `Libro`
- questo fenomeno è chiamato **polimorfismo** ed è uno dei principi fondamentali della programmazione orientata agli oggetti
- una variabile come `lib` definita nell'esempio precedente è **polimorfa**: può contenere oggetti di tipo diverso

- *in informatica, il termine polimorfismo (dal greco "avere molte forme") viene usato in senso generico per riferirsi a espressioni che possono rappresentare valori di diversi tipi (dette espressioni polimorfiche)*
- *nel contesto della programmazione orientata agli oggetti, si riferisce al fatto che una espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (polimorfismo per inclusione)*

*wikipedia*

- un oggetto della classe base non può essere utilizzato al posto di uno della classe derivata.
- nel nostro esempio una situazione come la seguente genera un errore:

```
Libro lib = new Libro();  
LibroDiTesto libtes;  
libtes = lib;          /*** ERRORE ***/
```



- l'ereditarietà può estendersi a più livelli generando una **gerarchia di classi**
- una classe derivata può, a sua volta, essere base di nuove sottoclassi
- nell'esempio: Sportivo è sottoclasse di Persona ed è superclasse di Nuotatore, Motociclista e Calciatore
- nella parte alta della gerarchia troviamo le **classi generiche**, scendendo aumenta il **livello di specializzazione**

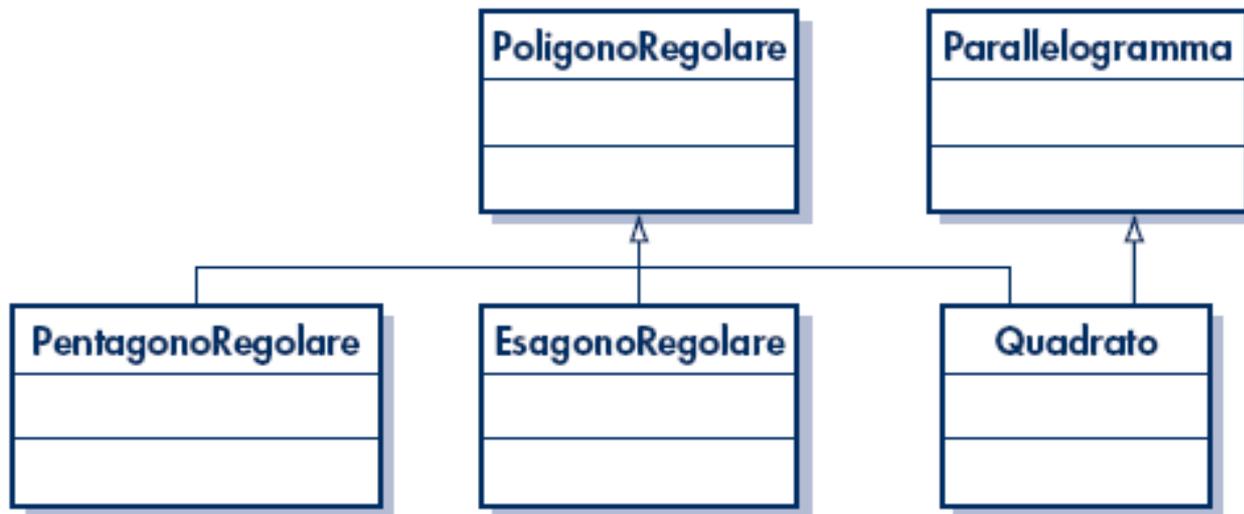
```
class Persona {  
    ...  
}  
  
class Sportivo extends Persona {  
    ...  
}  
  
class Nuotatore extends Sportivo {  
    ...  
}  
  
class Motociclista extends Sportivo {  
    ...  
}  
  
class Calciatore extends Sportivo {  
    ...  
}
```

# ereditarietà singola e multipla

A. Ferrari

- sono possibili due tipi di ereditarietà:
  - ereditarietà singola
  - ereditarietà multipla
- **l'ereditarietà singola** impone ad una sottoclasse di derivare da **una sola superclasse**
- l'esempio presentato precedentemente è un caso di ereditarietà singola: ogni sottoclasse ha una sola classe base, mentre è possibile da una superclasse avere più classi derivate
- vari linguaggi ad oggetti pongono il vincolo dell'ereditarietà singola per problemi di chiarezza e semplicità d'implementazione, **Java è uno di questi**
- **non** è possibile quindi una definizione di classe del tipo:  
**class A extends B,C**

- l'ereditarietà multipla si ha quando una sottoclasse deriva da **più** superclassi
- la classe **Quadrato** ha due superclassi: **PoligonoRegolare** e **Parallelogramma**



# Java e l'ereditarietà multipla

A. Ferrari

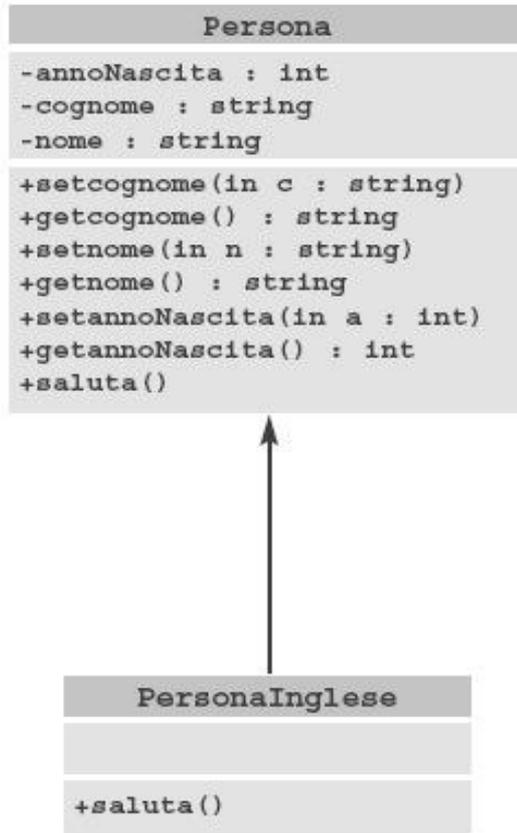
- Java **non** prevede ereditarietà multipla fra classi
- l'introduzione delle **Interfacce** (che vedremo in seguito) permette parzialmente di ovviare a questa limitazione

- una classe derivata può differenziarsi dalla classe base aggiungendo nuove caratteristiche:
  - nuovi attributi
  - e/o nuovi metodi
- in questo caso si parla di ***estensione***
- l'esempio relativo alla classe Libro e LibroDiTesto è un esempio di ereditarietà per estensione: la sottoclasse **aggiunge nuove caratteristiche** ma non altera il comportamento delle funzionalità offerte dalla classe base

- la classe derivata potrebbe però fornire le stesse caratteristiche della classe base differenziandosi invece per il **comportamento**
- si definisce **ereditarietà per ridefinizione (overriding)** la situazione in cui uno o più **metodi** della classe base siano **ridefiniti** nella classe derivata
- i metodi avranno quindi la **stessa firma** (nome e lista di tipi dei parametri) ma **differente corpo**

# un esempio di overriding

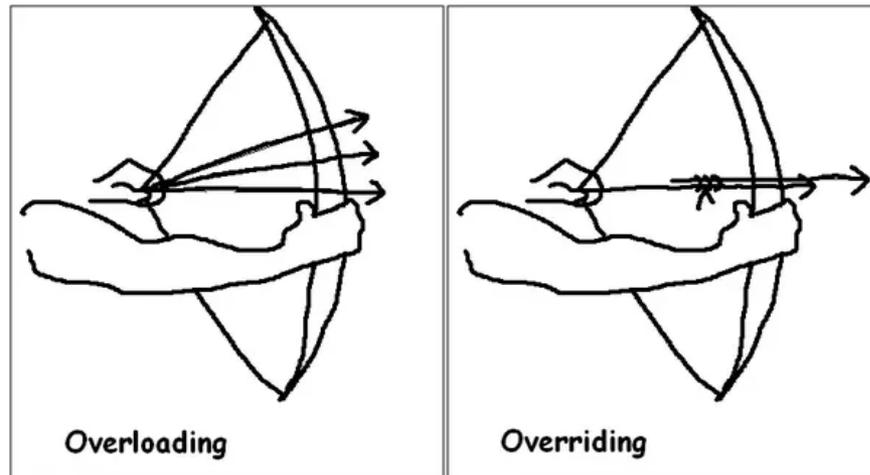
A. Ferrari



```
class Persona {
    ...
    ...
    public void saluta() {
        String saluto="Salve sono ";
        saluto+=cognome+" "+nome;
        System.out.print(saluto);
    }
    ...
    ...
}

class PersonaInglese extends Persona {
    public void saluta() {
        String saluto="Hallo I'm ";
        saluto+=cognome+" "+nome;
        System.out.print(saluto);
    }
}
```

- attenzione a non confondere
- il **sovraccarico** dei metodi (**overloading**)  
situazione in cui oltre al corpo del metodo è differente anche la sua firma
- con la **ridefinizione** (**overriding**)  
situazione in cui la firma del metodo è identica ma è differente il corpo



- è possibile incontrare situazioni in cui sono presenti **sia estensione che ridefinizione**
- nell'esempio presentato in precedenza relativo alla biblioteca scolastica avremmo potuto ridefinire nella classe LibroDiTesto il metodo stampa() per includere la visualizzazione dei nuovi attributi

- la parola chiave **this**, intesa come riferimento all'oggetto stesso, è utilizzata per eliminare ambiguità o per esplicitare ulteriormente il riferimento ad un attributo o ad un metodo interno
- in una gerarchia di classi può essere necessario far riferimento ad un **attributo o metodo della superclasse**, in questo caso si utilizza la parola chiave **super**

```
class Libro {
    ...
    public void stampa () {
        System.out.print (autore+" ");
        System.out.print (titolo+" ");
        System.out.print ("pag. "+numeroPagine);
        System.out.print (" "+codiceISBN);
    }
    ...
}

class LibroDiTesto extends Libro {
    ...
    public void stampa () {
        super.stampa ();
        System.out.print (" "+materia);
    }
    ...
}
```

- Java permette di trasformare il tipo di un oggetto mediante l'operazione di casting
- l'operatore di casting viene premesso al riferimento dell'oggetto:  
**(tipo) oggetto**
  - dove tipo è il nuovo tipo di oggetto (nome della classe)

- tutte le classi Java sono **implicitamente derivate dalla classe Object** che risulta quindi al vertice di ogni gerarchia di classi
- la classe Object implementa alcuni metodi che possono essere ridefiniti dalle nuove classi
- in particolare può risultare utile ridefinire il metodo **toString()** che fornisce una rappresentazione della classe sotto forma di stringa

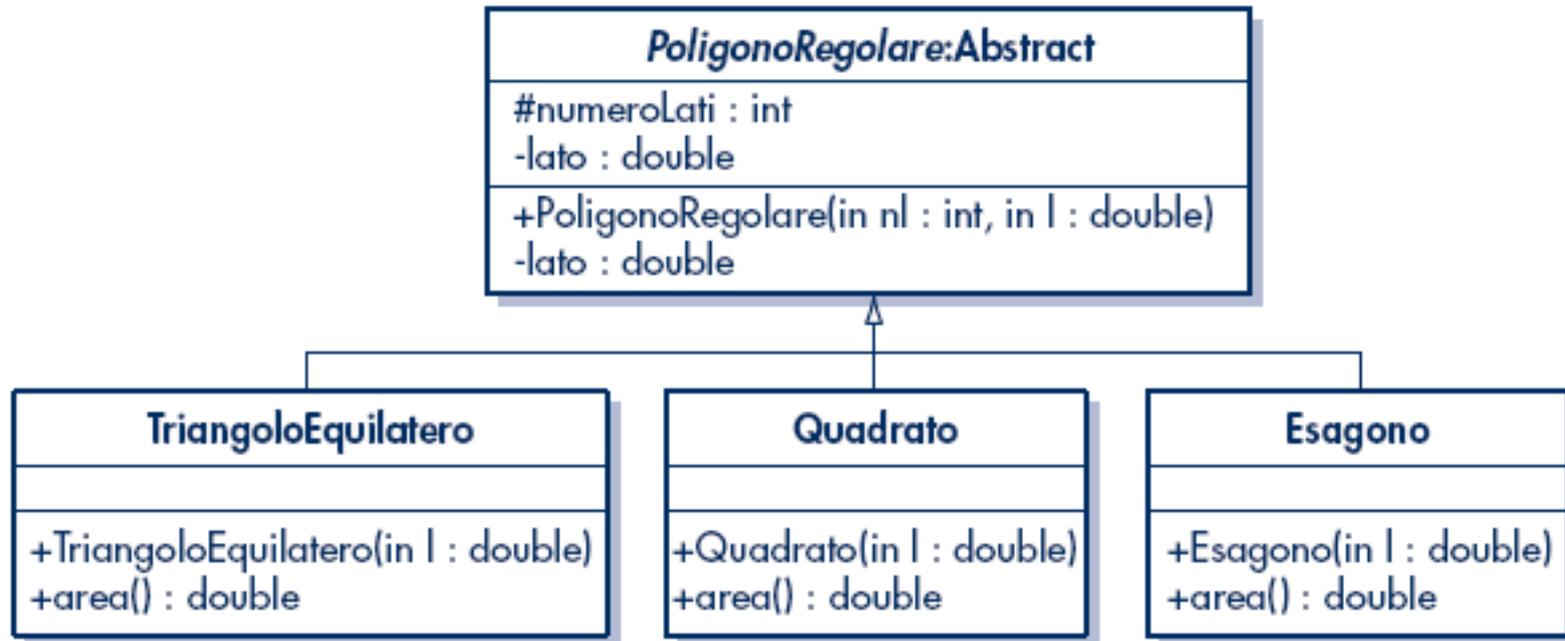
◆ equals (Object)	boolean
◆ getClass ()	Class
◆ hashCode ()	int
◆ notify ()	void
◆ notifyAll ()	void
◆ toString ()	String
◆ wait (long, int)	void
◆ wait (long)	void
◆ wait ()	void

- abbiamo fino ad ora utilizzato il modificatore **final** per definire **attributi non modificabili** (costanti)
- **final** può essere utilizzato anche **per i metodi**; anche in questo caso sta a significare la non modificabilità del metodo (*un metodo dichiarato final non può essere ridefinito da una sottoclasse*)
- esistono anche **classi** definite **final** che non possono essere estese (*una classe dichiarata final non può avere sottoclassi*)

```
class Parola extends String {  
    public boolean isUnSaluto() {  
        if (this.equals("ciao"))  
            return true;  
        else  
            return false;  
    }  
}
```

- nell'esempio si tenta di estendere la classe String aggiungendo un nuovo metodo
- l'esempio non è funzionante perché la classe String fornita dal linguaggio è definita final quindi non estendibile
- l'errore in fase di compilazione è il seguente:  
`Cannot inherit from final java.lang.String`

- si definiscono **metodi astratti** quei metodi in cui è presente **solo la firma ma non il corpo** (il modificatore di un metodo astratto è `abstract`)
- una classe che contiene almeno un metodo astratto si definisce **classe astratta** e da essa **non** possono essere istanziati oggetti
- le sottoclassi di una classe astratta devono implementare **tutti** i metodi astratti della classe base o essere a loro volta astratte



```

abstract class PoligonoRegolare {
    // Superclasse della gerarchia dei poligoni regolari
    protected int numeroLati; //numero lati poligono
    protected double lato; //lunghezza del lato
    // Costruttore
    PoligonoRegolare(int nl,double l) {
        numeroLati=nl;
        lato=l;
    }
    public double perimetro() {
        return lato*numeroLati;
    }
    abstract double area ();
}

class TriangoloEquilatero extends PoligonoRegolare {

    TriangoloEquilatero(double l) {
        // viene attivato il costruttore della superclasse
        super(3,l);
    }

    // implementazione del metodo astratto della superclasse
    public double area() {
        return (lato*altezza()/2);
    }
    // metodo privato per il calcolo dell'altezza del triangolo
    private double altezza () {
        return (lato*Math.sqrt(3)/2);
    }
}

```

- l'ereditarietà facilita il **riutilizzo** di software estendendone o ridefinendone caratteristiche e comportamenti; è possibile adattare una classe preesistente alle nuove esigenze
- specificare le differenze da una classe simile piuttosto che ridefinire completamente la classe facilita enormemente lo sviluppo di nuovi progetti poiché **elimina ridondanza di codice**
- l'ereditarietà non è un meccanismo di inclusione del codice di una classe base in una derivata.
  - **non c'è copia di codice**, ogni modifica della struttura di una classe base si ripercuote automaticamente nelle sue classi derivate

# l'interfaccia verso il mondo esterno

A. Ferrari

- l'*interfaccia* di un oggetto è l'insieme delle firme dei suoi metodi
- se non si vuole dare accesso diretto agli attributi, e se si vuole nascondere l'implementazione di una classe, l'unica cosa che deve conoscere chi utilizza la nostra classe, è l'interfaccia
- conoscere l'interfaccia significa sapere quali sono le operazioni (metodi) che si possono invocare su un oggetto

# definizione esplicita di interfaccia

A. Ferrari

- una interfaccia (interface) in Java ha una struttura simile a una classe, ma può contenere solo metodi astratti e costanti (quindi non può contenere costruttori e attributi)
- l'interfaccia non essendo una classe implementata non può essere istanziata direttamente
- una classe che vuole fare uso di un'interfaccia si dice che la implementa; questo rende obbligatorio l'implementazione di tutti i metodi definiti nell'interfaccia

- l'interfaccia fornisce uno **schema** di come dovrà essere strutturata la classe: quali metodi dovranno essere presenti
- l'interfaccia **non fornisce l'implementazione** dei metodi ma lascia allo sviluppatore l'onere di scriverli in modo specifico per ognuna delle classi che fa uso dell'interfaccia
- l'utilizzo di un'interfaccia è utile quando si devono definire i metodi che dovrà possedere un oggetto senza poterne dare un'implementazione univoca per tutte le tipologie di oggetti che faranno uso dell'interfaccia

```
public interface Risorsa {
    public String leggi();
    public void Scrivi(String buffer);
    public int disponibili();
}

public class RisorsaFile implements Risorsa {
    public String leggi() {
        // Ritorno il contenuto del file
    }
    public void Scrivi(String buffer) {
        // Scrivo il contenuto di buffer nel file
    }
    public int disponibili() {
        // Ritorno il numero di caratteri disponibili nel file
    }
}
```

# interfacce ed ereditarietà multipla

A. Ferrari

- una classe può implementare anche **più di una interfaccia**
- è possibile implementare un'**interfaccia** e contemporaneamente estendere una **classe**.
- in questo modo si può ovviare al limite di Java di non possedere l'**ereditarietà multipla**.
- rimane l'inconveniente che utilizzando un'interfaccia per simulare l'ereditarietà multipla è necessario **implementarne tutti i metodi**, rendendo le scelte di progetto difficili e delicate.