

OOP in Java



JavaTM

Classi astratte e
interfacce

Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia

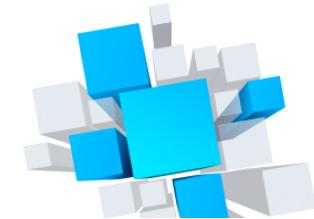


Classi astratte

Cosa significa "Classe Astratta"?

Una classe **astratta** è una classe che **non può essere istanziata** ma solo estesa (sono infatti definite anche classi **incomplete**)

Una classe astratta è caratterizzata dall'utilizzo della keyword **abstract**



JAVA

```
public abstract class <nomeclasse> {  
    // lista degli attributi  
    // metodi e costruttore  
}
```



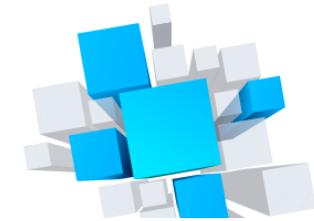
```
/**  
 * @param args the command line arguments  
 */  
public static void main(String[] args) {  
    // TODO code application  
    Astratta a = new Astratta();  
}
```



Classi Astratte

Anche i metodi possono essere definiti astratti se preceduti dal modificatore **abstract**

Le classi astratte hanno **generalmente**, ma non esclusivamente, dei metodi che sono anch'essi definiti **astratti** e che sono privi di un corpo di definizione: sono solo dichiarati, ma non forniscono alcuna implementazione.



Una classe è astratta anche se contiene **almeno un metodo astratto**

Una classe astratta può comunque essere derivata, e le classi che derivano da essa devono implementarne gli eventuali metodi astratti, sovrascrivendoli con una propria logica specifica.^[1]

Un metodo definito **astratto** è un metodo solo **dichiarato** ma non **implementato**. Tale metodo **deve** necessariamente essere implementato nelle classi derivate se si vuole che queste possano essere instanziate.

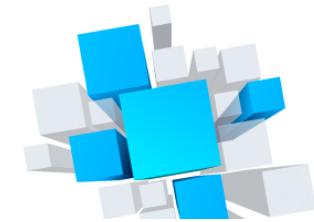
[1] Pellegrino Principe - Java 11 - Edizioni Apogeo



Classi Astratte

Di seguito elenchiamo alcuni punti importanti da ricordare per le classi astratte:

- una sottoclasse di una classe astratta deve obbligatoriamente implementarne gli eventuali metodi astratti e, se non vi provvede, essa stessa deve divenire classe astratta;
- una classe astratta può avere anche membri dati e metodi non astratti;
- una classe astratta non può essere final;
- genera un errore di compilazione creare oggetti di una classe astratta; si può solo creare una variabile del suo tipo, alla quale assegnare poi istanze delle sue sottoclassi.



Per i metodi astratti elenchiamo invece i seguenti punti da rammentare:

- la loro dichiarazione è permessa solo nelle classi astratte;
- da una classe derivata da una classe astratta non è possibile usare super per farvi riferimento; per esempio, se A è una classe astratta che dichiara il metodo astratto m e D è una sua classe derivata, da un metodo di D non potremo usare super.m().

Perché usare classi astratte?

Le classi astratte vengono create per impostare una applicazione su un **modello** di classe che di per sé non ha senso di esistere ma aiuta (e obbliga) il programmatore delle classi reali a rispettare alcune regole formali.

Spieghiamo con un esempio

- Si vuole progettare un sistema di calcolo dello stipendio per una azienda.
- Lo stipendio va calcolato in maniera diversa in funzione del ruolo e delle attività svolte da ogni persona che lavora per quella azienda.
- Non tutte le funzioni lavorative sono note al progettista al momento dell'implementazione del sistema (potrebbero nascerne di altre in futuro)



L'unica cosa certa è che ogni lavoratore avrà almeno un nome, un cognome, una mansione ed uno stipendio base legato alla mansione.

- Il progettista baserà l'implementazione sulle poche cose certe che conosce lasciando **astratto** il metodo di calcolo dello stipendio **obbligando** di fatto il progettista delle classi derivate a **implementare** tale metodo nel modo corretto.
- Il resto dell'applicazione potrà però essere formalmente implementata.

Un esempio...

Progettare la classe astratta Employee obbligherà chiunque voglia progettare le classi derivate a dover **implementare** il metodo **calcolaStipendio()** e **toString()** rispettando la firma dei metodi

```
public abstract class Employee {  
  
    private final String nome;  
    private final String cognome;  
    private String mansione;  
    private GregorianCalendar dataassunzione;  
  
    public Employee(String n, String c, String m, GregorianCalendar d) {  
        this.nome = n;  
        this.cognome = c;  
        this.mansione = m;  
        this.dataassunzione = d;  
    }  
  
    public abstract float calcolaStipendio();  
  
    @Override  
    public abstract String toString();  
  
    public final int giorniPermanenze() {  
        GregorianCalendar today = new GregorianCalendar();  
        return (int)((today.getTimeInMillis()-this.dataassunzione.getTimeInMillis())/1000/3600/24);  
    }  
}
```

JAVA



Un esempio... [2]

Nel frattempo si possono scrivere le funzioni applicative e le strutture dati che utilizzano la forma della classe astratta

```
public class CalcoloPaga {  
  
    public static void printSalary(ArrayList<Employee> employees) {  
  
        for (Employee e : employees) {  
            System.out.println(e);  
            System.out.println(e.calcolaStipendio());  
        }  
  
    }  
  
    public static void main(String[] args) {  
  
        ArrayList<Employee> employees = new ArrayList<>();  
  
        printSalary(employees);  
  
    }  
}
```

JAVA



Esercizio

Sia dato il file di testo **employees.csv** strutturato come segue che contiene i dati di tutti i lavoratori di una azienda

tipo	nome	cognome	mansione	stipendiobase	dataassunzione
IMPIEGATO	Robert	Zemeckis	segreteria	1500	2000-01-01
IMPIEGATO	Sandra	Bullock	gestionale	1650	2001-10-15
AMMINISTRATORE	Tom	Cruise	commerciale	2100	2005-06-01
AMMINISTRATORE	Glenn	Close	finanziario	2150	2003-02-15
TECNICO	Kevin	Costner	informatico	2006	2006-12-01
TECNICO	Jennifer	Aniston	sistemista	1850	2008-10-10

EMPLOYEES.CSV

A partire dallo schema precedente, implementare le **classi** corrette e scrivere un programma in Java che, letto il **file di testo**, produca in output lo stipendio di ogni persona calcolato anche in funzione del **tempo di permanenza** in azienda





Interfacce

Cosa significa "Interfaccia" ?

Una interfaccia in Java è una struttura simile a una classe ma può contenere **SOLO** metodi d'istanza astratti e costanti

- (quindi non può contenere costruttori, variabili statiche, variabili di istanza e metodi statici).

Una interfaccia java **NON** può essere **estesa** ma solo **implementata**

Una classe che **implementa** una interfaccia **DEVE** necessariamente implementare **TUTTI** i metodi presenti nell'interfaccia stessa



Perché le interfacce ?

Perché una nuova struttura se abbiamo già le classi astratte?

Si noti che:

- Una classe (astratta o concreta) può estendere solo una classe (astratta o concreta) ma **implementare più interfacce**
- Una classe (astratta o concreta) può essere usata per "fattorizzare" codice comune alle sue sottoclassi, mentre una interfaccia **non può contenere codice**
- Una classe astratta può contenere chiamate di metodi astratti prescindendo dalla loro implementazione, una classe concreta non può usare metodi astratti

Nell'uso delle interfacce in un programma, ricordarsi delle seguenti regole:

- Possiamo **dichiarare** una variabile indicando come tipo un'interfaccia:
- **Non possiamo istanziare** un'interfaccia:
- Ad una variabile di tipo interfaccia possiamo **assegnare** solo istanze di classi che implementano l'interfaccia:
- Su di una variabile di tipo interfaccia **possiamo invocare** solo metodi dichiarati nell'interfaccia (o nelle sue "super-interfacce").



Dichiarazione di interfaccia

La dichiarazione è simile a quella delle classi ma con la keyword **interface**

```
public interface <Int> [extends <Int1>, <Int2>, ...] {  
    <tipol> <var1>= <val1>;  
    ...  
  
    <tipoN> <varN> = <valN>;  
  
    <tipo-res1> <metodo1> ( <lista-parametri1> );  
    ...  
  
    <tipo-resM> <metodoM> ( <lista-parametriM> );  
}
```

JAVA

Nota che:

- Le variabili devono essere inizializzate e non possono essere modificate successivamente: anche se non sono dichiarate final di fatto sono delle **costanti**;
- I metodi sono tutti **astratti**: infatti al posto del corpo c'è solo un punto e virgola;
- I metodi dichiarati in una interfaccia sono sempre **public**. Di conseguenza, i corrispondenti metodi di una classe che implementa l'interfaccia devono essere public.
- Una interfaccia può estendere una o più interfacce (non classi), indicate dopo la parola chiave extends. Per le interfacce non vale la restrizione di **ereditarietà singola** che vale per le classi.



Uso di interfacce

Ogni classe estende (extends) una sola altra classe (Object se non specificata);

Una interfaccia può estendere (extends) una o più interfacce:

```
public interface <Int> extends <Int1>, <Int2>, ...{  
    ...  
}
```

JAVA

La gerarchia di ereditarietà singola delle classi e la gerarchia di ereditarietà multipla delle interfacce sono completamente disgiunte.

Una classe può implementare (implements) una o più interfacce:

```
public class <nomeClasse> extends <nomeSuperClasse> implements <Int1>, <Int2>, ..., <Intn> {  
    ...  
}
```

JAVA

Una interfaccia definisce un tipo di dati astratto, di cui fornisce la **specifica** delle operazioni

- la struttura dei suoi elementi e il modo in cui le operazioni sono effettivamente definite verrà determinato dal tipo di dati (la classe) che realizza (implements) l'interfaccia.



Esempio di uso delle interfacce

Progetto WebMarket

- Si vuole progettare un sistema per la vendita on line di prodotti **qualsiasi**
- Il sistema dovrà funzionare con prodotti diversi forniti da fornitori diversi che hanno i loro propri sistemi informativi

Come affrontare questo problema?

- Bisogna prevedere delle classi che possano **interagire** con oggetti di tipo diverso ma che devono rispettare almeno alcune **regole di funzionamento**
- Queste "regole" sono proprio quelle che in Java vengono definite **interfacce**"
- Assumiamo che per essere venduto un oggetto qualsiasi debba avere almeno alcune caratteristiche, ovvero debba essere **vendibile**, per esempio:
 - abbia un metodo per fornire un prezzo
 - abbia un metodo per restituire una descrizione
- La prima cosa che si può fare è proprio definire l'interfaccia di ogni oggetto verso il nostro sistema.

L'interfaccia è il **contratto** che ogni fornitore dovrà far rispettare al proprio oggetto affinché sia vendibile



Esempio: Progetto WebMarket

Prima ancora di scrivere il **Main** della nostra applicazione definiamo appunto il **contratto** da rispettare.

```
public interface Vendibile {  
    double getPrezzo();  
    String getDescrizione();  
}
```

JAVA

La struttura **interface** di Java vuole solo la **firma** dei metodi e non l'implementazione (come le classi astratte)

REGOLE

Non bisogna specificare **abstract** perché in una interfaccia **tutti** i metodi sono per definizione astratti

Non bisogna specificare **public|private|protected** perché in una interfaccia **tutti** i metodi sono per definizione pubblici

Non si possono indicare attributi di istanza (perché una interfaccia è un concetto astratto)

Si possono indicare attributi statici (non è il nostro caso)



Esempio: Progetto WebMarket

Definiamo adesso il comportamento di chi deve manipolare oggetti vendibili (creiamo una classe Venditore) prima ancora di capire chi saranno questi oggetti vendibili

```
public class Venditore {  
    String nome;  
    ArrayList<Vendibile> venduto;  
    double totalevenduto = 0;  
  
    public Venditore(String nome) {  
        this.nome = nome;  
        this.venduto = new ArrayList();  
    }  
  
    public void vendi(Vendibile v) {  
        this.venduto.add(v);  
        System.out.println("Ho venduto " + v.getDescrizione());  
        this.totalevenduto = this.totalevenduto + v.getPrezzo();  
    }  
}
```

JAVA

Il Venditore funziona indipendentemente dal tipo di oggetto che sta vendendo

fondamentale è che tale oggetto abbia un metodo `getPrezzo()` e un metodo `getDescrizione()`



Esempio: Progetto WebMarket

Definiamo adesso due classi di elementi da vendere. per esempio Penna e Quaderno

La definizione della classe **Penna** unitamente alla specificazione **implements** dell'interfaccia da implementare dà luogo al seguente errore

E' obbligatorio **implementare** tutti i metodi previsti dall'interfaccia che stiamo utilizzando

```
3  /**
4  *
5  * @author Giulio
6  */
7  public class Penna implements Vendibile {
8
9 }
10
```

A tooltip window is overlaid on the code, pointing to the word "implements". The text in the tooltip reads: "Penna is not abstract and does not override abstract method getDescrizione() in Vendibile" followed by a dashed line and "(Alt-Enter shows hints)".

Esempio: Progetto WebMarket

l'IDE ci crea tutte le implementazioni (senza un corpo reale).

```
public class Penna implements Vendibile {  
  
    @Override  
    public double getPrezzo() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
  
    @Override  
    public String getDescrizione() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
}
```

JAVA

Passiamo quindi a definire il comportamento di due classi concrete che implementano **Vendibile**



Esempio: Progetto WebMarket

```
public class Penna implements Vendibile {  
    @Override  
    public double getPrezzo() {  
        return 1.50;  
    }  
    @Override  
    public String getDescrizione() {  
        return "Penna a sfera blu";  
    }  
    public String getMarca() { // metodo solo di Penna  
        return "BIC";  
    }  
}
```

JAVA

```
public class Quaderno implements Vendibile {  
    @Override  
    public double getPrezzo() {  
        return 2.50;  
    }  
    @Override  
    public String getDescrizione() {  
        return "Quaderno a righe formato A4";  
    }  
    public int getNumeroPagine() { // metodo solo di Quaderno  
        return 20;  
    }  
}
```

JAVA



Esempio: Progetto WebMarket

Adesso proviamo a mettere insieme il tutto in un piccolo **Main** di test

```
// se provo a instanziare un oggetto di tipo "Vendibile" ricevo un errore
// perché Vendibile è un concetto astratto

// Vendibile o = new Vendibile(); // => "Vendibile is abstract; cannot be instantiated"

public static void main(String[ ] args) {
    Venditore v = new Venditore("Marco");
    Penna p = new Penna();
    Quaderno q = new Quaderno();
    v.vendi(p);
    v.vendi(q);
}
```

JAVA

```
Ho venduto Penna a sfera blu
Ho venduto Quaderno a righe formato A4
```

OUTPUT



Esempio: Progetto WebMarket

Fino a qui però nulla di nuovo rispetto alle classi astratte!

Perché tutto questa complessità aggiunta?

Definiamo ora una classe Acquirente, per modellare il sistema che si occupa di acquistare dai fornitori per rivendere on line.

In Java **non** è permessa l'ereditarietà multipla. Non si può definire una classe che ne estende più di una, mentre si può invece implementare più interfacce

Definiamo quindi il comportamento che deve avere un oggetto per poter essere **acquistato** dal sistema

```
public interface Acquistabile {  
    String getDescrizione();  
    double getPrezzoAcquisto();  
    int getQuantitaMinima();  
}
```

JAVA

Il nostro sistema è implementato partendo dal suddetto contratto.

ne segue che, chi vuole vendere tramite il nostro web market, deve prevedere **anche** questi metodi



Esempio: Progetto WebMarket

La classe Acquirente, definita nel nostro sistema, gestisce il magazzino tramite questa interfaccia

```
public class Acquirente {  
    ArrayList<Acquistabile> acquistato;  
  
    public Acquirente() {  
        this.acquistato = new ArrayList();  
    }  
    public void acquista(Acquistabile a, int numeromassimo) {  
        if (a.getQuantitaMinima() <= numeromassimo) {  
            this.acquistato.add(a);  
            System.out.println("Ho comprato " + a.getDescrizione() +  
                " al prezzo di " + a.getPrezzoAcquisto());  
        }  
        else {  
            System.out.println("Non posso comprare " + a.getDescrizione() +  
                " perché l'acquisto minimo supera la soglia di " + numeromassimo);  
        }  
    }  
}
```

JAVA



Esempio: Progetto WebMarket

Modifichiamo adesso il main per acquistare i prodotti che vogliamo vendere

Proviamo ad acquistare al massimo 10 penne...

```
// istruzioni di acquisto prodotti
Penna p = new Penna();
Acquirente a = new Acquirente();
a.acquista(p, 10); // => ERRORE incompatible types: Penna cannot be converted to Acquistabile
```

JAVA

Ecco quel qualcosa in più che ci consentono le **interface** di Java.

E' sufficiente che le classi **Penna** e **Quaderno** implemen~~tino anche~~ti l'altra interfaccia per poter essere utilizzate

```
public class Penna implements Vendibile, Acquistabile {
    // altri metodi di Penna
    @Override
    public double getPrezzoAcquisto() { return 0.9; }
    @Override
    public int getQuantitaMinima() { return 5; }
}
```

JAVA



Esempio: Progetto WebMarket

Il main non restituisce più errori

```
// istruzioni di acquisto prodotti  
Penna p = new Penna();  
Acquirente a = new Acquirente();  
a.acquista(p, 10);
```

JAVA

similmente in Quaderno

E' sufficiente che le classi **Penna** e **Quaderno** immplementino **anche** l'altra interfaccia per poter essere utilizzate

```
public class Quaderno implements Vendibile, Acquistabile {  
    // altri metodi di Penna  
    @Override  
    public double getPrezzoAcquisto() { return 1.5; }  
    @Override  
    public int getQuantitaMinima() { return 20; }  
}
```

JAVA



Esempio: Progetto WebMarket

Lanciando il main ottengo il seguente output...

```
public static void main(String[ ] args) {  
  
    Penna p = new Penna();  
    Quaderno q = new Quaderno();  
    Acquirente a = new Acquirente();  
    a.acquista(p, 10);  
    a.acquista(q, 5);  
  
    Venditore v = new Venditore("Marco");  
    v.vendi(p);  
    v.vendi(q);  
  
}
```

JAVA

```
Ho comprato Penna a sfera blu al prezzo di 0.9  
Non posso comprare Quaderno a righe formato A4 perché l'acquisto minimo supera la soglia di 5  
Ho venduto Penna a sfera blu  
Ho venduto Quaderno a righe formato A4
```

OUTPUT



Esercizio: Progetto WebMarket

Si vuole progettare un sistema completo di gestione aziendale.

Scrivere un programma JAVA che gestisca un insieme di elementi "vendibili"

Ogni elemento deve avere **necessariamente** i metodi

- `getDescrizione() => string`
- `getDescrizioneEng() => string`
- `setDescrizione(string) => string`
- `setDescrizioneEng(string) => string`
- `getPrice() => double`
- `setPrice(double) => void`



Esercizio: Progetto WebMarket

Gli elementi vendibili sono gestiti tramite le interfacce **Saleable** e **Purchasable**

La lista degli elementi presente nel file di testo "elementi.csv" che ha il seguente formato:

```
codice,tipo,descrizione,prezzounitario,quantitamagazzino  
PEN001,Penna,Penna a sfera,1.50,1545  
MOU002,Mouse,Mouse ottico,9.99,897  
...
```

TESTO

Prevedere una classe Gestore che contiene l'elenco di tutti gli elementi disponibili con relativa quantità disponibile (deve leggere e interpretare i dati presenti nel file)



Esercizio: Progetto WebMarket

Gestore espone i servizi

- vendi(Saleable s, int n)
 - vende n elementi s
 - nota: dà errore se ce ne sono in magazzino meno di n
- disponibile(Saleable s) => int
 - restituisce il numero di elementi s presenti in magazzino
- magazzino() => array di elementi ognuno con la relativa disponibilità
 - suggerimento: HashMap
- carica(Purchasable p, int n) => void
 - carica in magazzino n elementi di tipo p

Suggerimento:

Non potendo creare una classe per ogni possibile tipo di elemento si può pensare ad usare il meccanismo dell'ereditarietà...



Esercizio: Progetto WebMarket(facile)

Sia dato il file di testo "elementi.csv" che ha il seguente formato:

TESTO
codice,tipo,descrizione,prezzounitario,quantitamagazzino
PEN001,Penna,Penna a sfera,1.50,1545
MOU002,Mouse,Mouse ottico,9.99,897

Nel file sono presenti elementi di vario tipo ma tutti Articoli

La classe astratta **Articoli** deve esporre i metodi:

- mostraPrezzo() => double // prezzo
- magazzino() => int // numero elementi presenti
- vendi(int n) // decrementa il numero presente in magazzino di n
- getDescrizione() => string // mostra descrizione

Dopo aver creato le strutture dati necessarie, leggere il file di testo e riempire una struttura dati ArrayList

- nel main provare tutti i metodi (specialmente il funzionamento di **vendi**)
- scrivere una funzione **StampaMagazzino(ArrayList)** che stampa (su file) la lista degli articoli presenti, con relativa descrizione e prezzo unitario e in fondo stampi il valore del magazzino





Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia