PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI (OOP)

- Paradigma di programmazione nato negli anni '80 che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi

- Non è l'unico paradigma:
  - programmazione procedurale (Fortran, COBOL)
  - programmazione strutturata (Pascal, Ada, C)
  - programmazione funzionale (Lisp, Haskell)
  - …

- Java non è l'unico esempio di linguaggio object-oriented: lo sono anche Smalltalk, Eiffel, C++, C#, Python, Ruby, ecc.

CLASSI E OGGETTI

- La differenza tra classe e oggetto è la stessa differenza che c'è tra tipo di dato e dato
- La classe è la descrizione astratta di un tipo di dato: specifica cioè i metodi e gli attributi che quel tipo di dato possiede
- L'oggetto è una istanza della classe: quando si istanzia una "variabile" definendola di una certa classe, si crea un oggetto di quella classe rappresentato dal nome della "variabile" istanziata

ESEMPIO

- La classe `Melody` permetterà di descrivere astrattamente una qualsiasi melodia, ad esempio prevedendo al proprio interno
  - un array di note `pitches` e una stringa `title` (attributi)
  - le funzioni `contaNote`, `calcolaMaxPitch`, `trasponi` e `stampa` (metodi)

- Definita la classe, posso istanziare 2 oggetti di tale classe chiamati `miaMelodia` e `jingleBells`, ciascuno dei quali avrà uno stato differente (sequenza di note e autore differenti) e su cui si potranno invocare i metodi previsti

DICHIARAZIONE DI CLASSI

- Dichiarazione minimale

```
class MiaClasse {
    // dichiarazione di attributi, metodi e
        costruttori
}
```

- Il nome della classe per convenzione inizia con una lettera in maiuscolo
- Il corpo della classe contiene tutto il codice che serve per il ciclo di vita degli oggetti di tale classe.

DICHIARAZIONE DI CLASSI

```
class MiaClasse {
   // dichiarazione di attributi, metodi e costruttori
}
```

- Attributi: campi (variabili o costanti) che definiscono lo stato dell'oggetto
- Metodi: funzioni che implementano i comportamenti ammissibili dell'oggetto o che ne riflettono lo stato

- Attributi e metodi sono entrambi membri di una classe
- Costruttori: metodi speciali richiamati quando si istanzia un oggetto della classe per inizializzarne lo stato
  - Ad esempio per allocare spazio in memoria e per assegnare valori di default alle variabili

ESEMPIO

```
public class Note {
    public int pitch;
    private int octave;

    public Note() {
        pitch = 0;
        octave = 0;
    }

    public int getOctave() {
        return octave;
    }
}
```

DICHIARAZIONE DI CLASSI

- Quando si progetta una classe, si decide quali metodi e quali attributi essa contiene: queste entità saranno poi richiamabili dall'esterno della classe (con le debite limitazioni)

- Tale modo di operare trasforma attributi e metodi di una classe nella sua interfaccia verso l'esterno: si può pensare all'oggetto come ad una scatola nera di cui si vede solamente l'interfaccia. L'interfaccia nasconde all'esterno l'implementazione interna

- Dall'esterno è noto "cosa fa" un metodo, ma non "come lo fa". I dati interni si possono dunque occultare e rendere accessibili attraverso metodi. Allo scopo si utilizzano i modificatori d'accesso

MODIFICATORI DI ACCESSO

- I modificatori di accesso sono parole chiave che regolano essenzialmente la visibilità e l'accesso ad un componente Java

- I membri di una classe si possono definire:
  - public: visibilità totale dall'interno e dall'esterno della classe (attributi accessibili in lettura e scrittura e metodi invocabili da chiunque)
  - private: visibilità ristretta alla classe proprietaria (attributi accessibili e metodi invocabili solo dall'interno della classe)
  - (altri modificatori d'accesso che verranno introdotti più avanti)

- Quando si progetta una classe è necessario individuare ciò che dovrà essere visibile all'esterno e cosa invece è necessario solo al corretto funzionamento interno: nel secondo caso è pericoloso permettere che altre classi possano modificare attributi o utilizzare funzioni pensate per lavorare sullo stato interno della classe stessa

INCAPSULAMENTO

I modificatori di accesso permettono di implementare la proprietà dell'incapsulamento, che ha due principali finalità:

1. Nascondere l'implementazione interna: dall'esterno della classe è noto cosa fa un metodo pubblico, cioè se ne conosce l'interfaccia (funzionalità svolta, parametri in ingresso e tipo restituito), ma non è noto come lo fa

2. Consentire un accesso protetto e "ragionato" dall'esterno:
   - ad esempio è possibile proteggere da scrittura un attributo definendolo private e implementando un metodo pubblico che ne restituisce il valore. Sarà possibile modificare tale valore solo dall'interno della classe;
   - ad esempio è possibile consentire di accedere in scrittura a un attributo non in modo diretto (public), ma definendolo private e creando un metodo public che effettua un controllo sui valori e gestisce eventuali situazioni di errore

COSTRUTTORI

- Quando un oggetto viene istanziato, cioè quando il compilatore trova la parola riservata new, invoca automaticamente un metodo particolare dell'oggetto: il costruttore

- Tale metodo è definito secondo alcune regole ben precise:
  - è un membro public della classe
  - ha lo stesso nome della classe
  - non restituisce alcun valore

```
public Nome_Classe([parametri]) { // corpo del metodo }
```

- E' possibile definire più costruttori (overload del costruttore), che devono distinguersi per numero e tipo di parametri passati.
  - Quando ha senso passare parametri a un costruttore? Ad esempio per inizializzare gli attributi con valori passati all'atto della costruzione dell'oggetto

IL COSTRUTTORE DI DEFAULT

- Nel caso non venga esplicitamente implementato, il compilatore utilizza un costruttore di default. Il costruttore infatti deve esistere comunque in una classe, a prescindere dalla esplicita volontà del progettista

```
Nota miaNota = new Nota(); // richiama il costruttore di default
```

- Il costruttore di default non ha alcun parametro di costruzione: eventuali attributi dell'oggetto non inizializzati rimangono tali fino ad una necessaria inizializzazione esplicita successiva

ISTANZIARE GLI OGGETTI

- Una volta definita una classe, un oggetto di tale classe si istanzia usando il nome della classe come se fosse un tipo di variabile e richiamandone il costruttore

```
Nome_Classe Nome_Istanza = new Nome_Classe();
```

- Nell'esempio di prima, richiamando il costruttore di default:

```
Melody jingleBells = new Melody();
```

IL COSTRUTTORE DI DEFAULT

- Attenzione: se viene esplicitamente definito anche un solo costruttore, allora il costruttore di default non viene costruito. Ad esempio:

```
public class MyClass {
      int k = -1;

      public MyClass(int i) {
            k = i;
      }
}
```

- Finché non viene implementato il costruttore in grassetto la seguente creazione di oggetto funziona correttamente, perché viene usato il costruttore di default; con l'implementazione in grassetto questo non è più consentito:

```
MyClass c = new MyClass();
```

ACCESSO AI MEMBRI DI UNA CLASSE

- Una volta dichiarati i membri di una classe, è possibile accedere ad essi utilizzando il carattere "."

```
Nome_Oggetto.Nome_Membro
```

- Risulteranno visibili all'esterno solo i membri dichiarati public. Nel caso di tentativo di accesso a un membro private dall'esterno della classe, il compilatore restituisce errore

- Esempio (vedi classe Note dichiarata in precedenza)

```
note.pitch
note.octave → Errore: l'attributo è private
note.getOctave()
```

ESEMPI

- L'istruzione Java
  ```
  System.out.println("Hello World");
  ```
  chiama il metodo println dell'oggetto System.out, che è fornito nella libreria standard del Java (JDK). Questo metodo stampa a video la stringa che riceve in ingresso

- Tale metodo però è proprio dell'oggetto System.out e se proviamo a richiamarlo su un oggetto diverso che non lo contempla, il compilatore restituisce errore:

  ```
  MyNoteClass miaNota = new MyNoteClass();
  miaNota.println("Hello World!!!"); // chiamata non valida
  ```

  La classe MyNoteClass di cui miaNota è un'istanza infatti non prevede un metodo pubblico println().

IL MODIFICATORE STATIC

- Applicabile a metodi e attributi di una classe
- Dichiarare un metodo static lo rende di fatto comune a tutte le istanze della classe. La chiamata del metodo avviene senza istanziare la classe:
  - Anziché `nomeOggetto.nomeMetodo()` si usa `nomeClasse.nomeMetodo()`

- Dichiarare un attributo static lo rende comune a tutte le istanze della classe
  - Un attributo public static è di fatto una variabile globale

PARAMETRO IMPLICITO THIS

- Dall'interno di un metodo è possibile accedere implicitamente agli attributi e metodi della classe di cui fa parte la funzione stessa
- Nel caso (sempre evitabile con opportune convenzioni di naming) che una variabile locale nasconda una variabile globale della classe, è necessario poter distinguere
- In ogni funzione, oltre ai parametri passati in ingresso, esiste sempre una variabile che fa riferimento alla classe stessa: è il parametro implicito `this`
- Utilizzando tale parametro e la notazione puntata è possibile disambiguare una variabile locale con lo stesso nome di una globale

UN HELLOWORLD AD OGGETTI

```
public class Messaggio {
	private String toPrint;

	public Messaggio(String print) {
		toPrint = print;
	}

	public void stampa() {
		System.out.println(toPrint);
	}

	public static void main(String args[]) {
		Messaggio ciaoMondo =
			new Messaggio("Hello World!");
		ciaoMondo.stampa();
	}
}
```

ENUMERAZIONI

ENUMERAZIONI

- Le enumerazioni (Enum o Enumerated Type) sono un **tipo** disponibile a partire dalla versione Java 5
- Le enumerazioni vengono introdotte dalla parola riservata enum
- Permettono di definire un insieme di valori predefiniti che una variabile (di quel tipo enumerativo) può assumere
- I valori sono costanti alfanumeriche, quali DO o C4

VANTAGGI E SVANTAGGI RISPETTO AI PRECEDENTI APPROCCI

- Le enumerazioni limitano il novero di possibili valori
  - Ad esempio, dichiarando il nome delle note come char, non era possibile inibire un'istruzione quale `char nota = 'W'`
- Le enumerazioni consentono di stabilire valori mnemonicamente facili da ricordare
  - Ad esempio, possiamo chiamare le note DO, RE, MI, ecc.
- Le enumerazioni non consentono operazioni aritmetiche
  - Ad esempio, non è banale selezionare il valore successivo nella sequenza di valori («cosa segue il SOL?»)

ENUMERAZIONI E CLASSI

- Una enumerazione è una classe, in particolare l'estensione della classe java.lang.Enum
- I tipi definiti in una enumerazione sono istanze di classe
- I valori di una enumerazione sono implicitamente public final static, quindi immutabili
- Il metodo == è sovrascritto, quindi può essere usato in maniera intercambiabile al metodo equals

SINTASSI

- Definizione del tipo enumerativo:

```
public enum nome_enum {val1, val2, ..., valN};
public enum Notes {C, D, E, F, G, A, B};
```

- Dichiarazione di variabili e assegnamento:

```
nome_enum nome_var = nome_enum.vali;
Notes miaNota = Notes.F;
Notes miaNota2 = miaNota;
```

- Operatori di confronto: == e !=
  - Non sono ammessi >, <, ecc.
  - Esiste il metodo compareTo che confronta i valori in base alla posizione nella dichiarazione (`a.compareTo(b)` è > 0 se pos(a) > pos(b))

- Conversione del valore in stringa: automatica

```
System.out.println(miaNota);
```

OPERAZIONI SUI VALORI

- Array dei valori contenuti nell'enumerazione:

  nome_enum.values()

- Tale array può essere trattato come qualsiasi altro. Ad esempio, ha una proprietà length che restituisce il numero di elementi nell'array. Accesso all'elemento n-esimo: nome_enum.values()[n]

- Recupero della posizione di un dato valore nella enumerazione, partendo da 0 rispetto alla dichiarazione: valore.ordinal()

OSSERVAZIONI

- Le enumerazioni sono efficaci quando:
  - i loro possibili valori appartengono a una singola tassonomia, e non sono aggregati. Anziché valori aggregati, ha più senso utilizzare più enumerazioni «mirate»
  - i loro possibili valori non trovano una rappresentazione numerica efficace, che rende invece possibile operazioni aritmetiche e di confronto

- Ad esempio, i nomi delle note sono una possibile enumerazione di 7 valori, e gli stati di alterazione un'altra di 5 valori. La combinatoria porterebbe invece ad avere $7 \cdot 5$ valori per una sola enumerazione e non ci sarebbero corrispondenze ad esempio tra note dello stesso nome ma con alterazioni diverse

- Ad esempio, ha poco senso codificare il numero di ottava di una nota attraverso un'enumerazione: meglio utilizzare i valori interi

ESEMPI

AlteredNotes.java

- Il software esemplifica il concetto di enumerazione e produce in uscita le 35 denominazioni delle 7 note, ciascuna con i 5 stati di alterazione

PitchCompare.java

- Il software legge in ingresso due nomi di nota, li converte nell'opportuno valore dell'enumerazione e restituisce informazioni su quale sia il pitch più acuto (non vengono considerate le alterazioni)

ESERCIZI

PitchTranspose.java

- Si scriva un programma che codifica il pitch della nota attraverso un'enumerazione, supportando come possibili valori quelli della notazione anglosassone
- Si scriva un metodo che riceve in input:
  - una variabile di tipo enumerativo
  - il numero di gradi di trasposizione in senso ascendente o discendente tramite un valore intero con segno
- …e restituisce in output la nota trasposta

DurationEnum.java

- Si scriva un programma che codifica le durate delle note attraverso un'enumerazione (es.: INTERO, META, QUARTO, ecc.), e che sfrutti la posizione dei valori nell'enumerazione per ricavare automaticamente le durate in forma frazionaria (es.: se la durata ha indice 2, si ottiene 1/4).

ESEMPIO
(Fraction.java)

Il codice contiene l'implementazione di una classe definita dall'utente per rappresentare i numeri razionali come frazione tra un numeratore e un denominatore intero. Esso contiene:

- diversi costruttori per inizializzare le strutture dati
- metodi ad hoc per effettuare le quattro operazioni aritmetiche elementari sulle frazioni
- un metodo per semplificare le frazioni, basato sul calcolo del massimo comun divisore
- un metodo per la rappresentazione in formato stringa delle frazioni

ESERCIZIO

(MusicFraction.java)

Si riveda il codice relativo alla classe Fractions allo scopo di rappresentare i valori ritmici delle note. In particolare:

- si aggiungano due enumerazioni che contengono le definizioni testuali delle durate (es.: INTERO, META, QUARTO, OTTAVO,… e SEMIBREVE, SEMIMINIMA, MINIMA, CROMA,…)
- si implementino dei costruttori che prendono in ingresso un valore di una delle enumerazioni per compilare opportunamente numeratore e denominatore
- si rivedano i costruttori che permettono di passare il valore del denominatore, passando l'esponente cui elevare la base 2 anziché il valore esplicito (ad esempio, per il denominatore di ½ si deve passare 1 in luogo di 2, per ¼ si deve passare 2 in luogo di 4, ecc.). Si faccia in modo che il denominatore sia compreso tra 1 e 8, quindi il valore minimo supportato sia il duecentocinquantaseiesimo

EREDITARIETÀ

- Ereditarietà: permette di derivare nuove classi (sottoclasse o classe figlia) a partire da quelle già definite (classi base, superclassi o classi padre) realizzando una gerarchia di classi

- Una sottoclasse:
  - mantiene i metodi e gli attributi delle classi base
  - può definire i propri metodi o attributi
  - può ridefinire alcuni dei metodi ereditati tramite un meccanismo chiamato overriding

- Una sottoclasse è più specializzata rispetto alla sua superclasse; analogamente, una superclasse è più generica rispetto a una sottoclasse

- Java supporta solo l'ereditarietà singola: un figlio può avere un solo padre → la gerarchia tra classi forma un albero

ESEMPIO DI EREDITARIETÀ

- Si consideri il rapporto gerarchico tra la superclasse MusicSymbol e la sottoclasse Note:
  - l'attributo positionX potrà essere comune a tutti i simboli musicali
  - Il metodo getPrintX, che restituisce la posizione orizzontale rispetto ad una certa scala di stampa, potrà essere definito nella superclasse, e rimarrà valido per tutte le sottoclassi che ereditano
  - una nota potrà avere attributi più specifici rispetto a un generico simbolo musicale, ad es. l'altezza o la durata
  - una nota potrà avere dei metodi più specifici; ad es. transpose(), che riceve in ingresso dei parametri e traspone la nota

DICHIARAZIONE DI SOTTOCLASSI

- Per dichiarare una sottoclasse Note che eredita da MusicSymbol si usa la sintassi:

```
class Note extends MusicSymbols{
   …
}
```

- In Java tutte le classi ereditano dalla superclasse Object, che definisce anche metodi comuni a tutte le sottoclassi (ad es. `equals()` e `toString()`)

ASSEGNAMENTI TRA OGGETTI

- L'operatore di assegnamento funziona tra oggetti della stessa classe

- Poichè una superclasse è più generica di una sottoclasse:
  - è sempre possibile assegnare un oggetto di una sottoclasse a un oggetto di una superclasse:
    ```
    superclasse = sottoclasse; // OK
    ```
  - non è possibile (e genera errore a compile-time) assegnare un oggetto di una superclasse a un oggetto di una sottoclasse:
    ```
    sottoclasse = superclasse ; // NON E' POSSIBILE!!!
    ```

- Esempio

```
MusicSymbol ms1 = new MusicSymbol();
MusicSymbol ms2 = new MusicSymbol();
ms1 = ms2; // ok
Note n = new Note(); // Note eredita da MusicSymbol
ms1 = n; // ok
n = ms1; // no!!!
```

MODIFICATORE D'ACCESSO PROTECTED E DEFAULT

- protected può essere attribuito solo a metodi e variabili interne ad una classe e non può essere applicato direttamente ad una classe

- I metodi e le variabili dichiarate come protected sono visibili da classi dichiarate nello stesso package e da sottoclassi e classi derivate dichiarate ovunque

- Se non viene specificato nulla, il compilatore considera un modificatore d'accesso detto di default.
  - Un membro di una classe (metodo o attributo) per default sarà accessibile solo dalle classi appartenenti al package dove è definito
  - Una classe appartenente ad un package definita senza il modificatore public sarà visibile solo dalle classi appartenenti allo stesso package

TABELLA RIASSUNTIVA MODIFICATORI D'ACCESSO

- I modificatori di accesso sono 3: private, protected e public
- In assenza di questi tre, un elemento di programma viene considerato package-local o friend (si dice che assume la visibilità di default)
- Le visibilità possibili sono:

Modificatore	Stessa classe	Stesso package	Sottoclasse	Ovunque
Public	sì	sì	sì	sì
Protected	sì	sì	sì	no
(default)	sì	sì	no	no
Private	sì	no	no	no

POLIMORFISMO E OVERRIDE

- Una classe che eredita può volere riscrivere e/o specializzare metodi ereditati. Nella programmazione ad oggetti l'override è l'operazione di riscrittura di un metodo ereditato

- Il metodo che effettua override deve avere lo stesso nome, stesso numero e tipo di parametri, e stesso tipo restituito rispetto a quello originario

- Polimorfismo (dal greco "avere molte forme"): un'interfaccia, molte implementazioni. I metodi che vengono ridefiniti in una sottoclasse sono detti *polimorfi*, in quanto lo stesso metodo si comporta diversamente a seconda del tipo di oggetto su cui è invocato

POLIMORFISMO DEI METODI: OVERLOAD E OVERRIDE

- Dal punto di vista implementativo il polimorfismo per i metodi si ottiene utilizzando l'overload e l'override dei metodi stessi

- Ogni metodo è identificato dal nome, ma è univocamente determinato anche dalla lista dei parametri. Ciò permette di avere all'interno di una classe più metodi che hanno lo stesso nome, ma con parametri diversi

- L'overload si basa sulla scrittura di più metodi identificati dallo stesso nome che però hanno, in ingresso, parametri di tipo e in numero diverso

- Con il termine override si intende una vera e propria riscrittura di un certo metodo di una classe che si eredita. Dunque, necessariamente, l'override implica ereditarietà

IL MODIFICATORE FINAL

- Il modificatore final, al pari di public e private, può essere preposto a un metodo, a un attributo o ad una classe:
  - preposto a una variabile la rende una costante per l'istanza della classe
  - preposto a un metodo, implica che in un contesto di ereditarietà su di esso non potrà essere eseguito l'override da parte della classe che eredita
  - una classe definita final non può essere ereditata. Questo semanticamente significa che la classe non necessita di specializzazioni o estensioni

INTERFACCE

- Un'interfaccia è una dichiarazione del "contratto" a cui devono sottostare le classi che la implementano
- Si dichiarano in modo simile alle classi, ma possono contenere solo costanti e dichiarazioni di metodi (e poco altro)
- Non si possono istanziare, ma solo implementare o estendere
- Esempio di dichiarazione:

```
public interface Melody {
     void transpose(int n);
     void retro();
}
```

IMPLEMENTAZIONE DI INTERFACCE

- Una classe può implementare una o più interfacce, usando la parola riservata implements. Ad esempio:

```
public class MyMelody implements Melody {
        public void transpose(int n) {
        …
        }

        public void retro() {
        …
        }
}
```

ESTENDERE LE INTERFACCE

- Un'interfaccia può estendere altre interfacce, con la parola riservata extends. Ad esempio:

```
public interface MelodyWithDuration extends Melody {
      int getTotalDuration();
}
```

- Attenzione:
  - una classe può ereditare (extends) una sola classe, ma può implementare (implements) diverse interfacce
  - un'interfaccia può ereditare (extends) da più interfacce

PAROLA CHIAVE ABSTRACT

- Una classe può essere definita con la parola chiave abstract, indicando che può contenere metodi astratti, ovvero senza la corrispondente implementazione
- Una classe astratta non può essere istanziata, ma può fungere da superclasse
- Esempio di dichiarazione:

```
public abstract class Song {
    public String title = "";
    abstract void play();
}
```

- Un'interfaccia può essere vista come una classe completamente astratta

DIFFERENZE TRA INTERFACCE E CLASSI ASTRATTE

- In una classe astratta ci possono essere attributi e metodi con relativa implementazione, mentre in un'interfaccia solo costanti e metodi senza implementazione

- Nelle interfacce, tutti i metodi dichiarati sono implicitamente public

- Una classe può ereditare da una sola altra classe (anche astratta), mentre può implementare diverse interfacce

ESEMPIO

(Note.java)

Il codice implementa l'ereditarietà tra una super-classe NoteRest (che rappresenta gli aspetti ritmici di un simbolo musicale, comuni a note e pause) e una sotto-classe Note, che la specializza con l'informazione sull'altezza e con opportuni metodi

ESERCIZIO

(TestCbr.java)

- Si implementi una classe che descrive singole note, distinta dalla classe principale che contiene il metodo main del software
- Per quanto riguarda l'altezza, si utilizzi internamente la Continuous Binomial Representation (CBR), esponendo metodi che permettono di leggere e scrivere l'informazione di ottava, pitch class e name class
- Si implementi un costruttore che accetta in input ottava, pitch class e name class
- Si protegga opportunamente l'accesso alla codifica interna dell'altezza, impedendo al metodo main di assegnare valori CBR inesistenti alle istanze delle note