

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

object-oriented programming OOP

modo di scrivere codice attraverso OGGETTI in grado di interagire tra di loro utile quando ci sono delle relazioni di indipendenza:

- oggetto Libro = composto da molti oggetti Capitolo, a loro volta composti da molti oggetti di tipo Pagina (composizione)
- Automobile e' un oggetto specifico di tipo Veicolo (specializzazione)
- un oggetto LettoreDVD necessita di un oggetto DVD (utilizzo)

permette di gestire piu' facilmente progetti di grandi dimensioni e rende il codice modulare e RIUSABILE

CLASSE

E' un tipo di dato astratto rappresentante un elemento (oggetto) con determinate caratteristiche (attributi) e operazioni (metodi)

Una classe deve rappresentare un singolo concetto = **coesione**

ad esempio una classe non puo' sia essere un registratore di cassa sia occuparsi del valore delle monete. In questo caso si devono creare due classi: una CashRegister e una Coin.

I nomi delle variabili e delle classi devono adeguarsi al loro uso
= **consistenza**, ovvero i nomi devono essere significativi e descrittivi per aiutare a capire cosa fanno. Inoltre i nomi devono attenersi alle Java Code Convention:

- nomi classi → PascalCase, lettera maiuscola 'Persona'
- attributi → camelCase, lettera minuscola 'longVariable'
- attributi costanti → tutto in maiuscolo 'MAX_HEIGHT'

METODI

Un **metodo accessore** e' un metodo che chiede all'oggetto di svolgere delle operazioni senza modificare lo stato interno dell'oggetto

es. classe Rectangle metodo getArea()

Un **metodo mutatore** modifica lo stato dell'oggetto ritornando void

es. setVariable()

Una classe immutabile è una classe senza alcun metodo mutatore
es. class String

MODIFICATORI DI ACCESSO

Determinano la possibilità di accedere a metodi e attributi di una classe da parte di altre classi

Ne esistono di quattro tipi:

- **private**, non sono visibili all'esterno della classe che li contiene
- **public**, sono visibili in tutte le classi, è "pubblico"
- **protected**, sono visibili nella classe stessa, nelle sottoclassi di quest'ultima e nelle classi che appartengono allo stesso package
- **default**, viene assegnato automaticamente quando si omettono altri modificatori e sono visibili nella classe stessa e in quelle appartenenti allo stesso package

Esistono altri due modificatori:

- **final**, usato su un attributo, esso diventa una costante e non può essere cambiato. Se usato su un metodo esso non può essere sovrascritto nelle sottoclassi. Se usato su una classe non potrà avere sottoclassi
- **static**, se lo applichiamo su un metodo lo rendiamo comune a tutte le istanze della classe. Applicato su un attributo, esso sarà condiviso da tutte le istanze di una classe, definendo così una variabile globale

EREDITARIETÀ

L'ereditarietà è uno dei meccanismi principali della programmazione ad oggetti. Grazie ad essa è possibile creare una **sottoclasse** (o classe derivata) che eredita tutti i metodi e gli attributi della **superclasse** (o classe base) che sta estendendo.

La sottoclasse eredita questi potendo poi aggiungerne nuovi oppure sovrascrivere alcuni ereditati (override). La classe base modella un concetto generico (es. class Person) e la classe derivata modella un concetto più specifico (es. class Student).

La sottoclasse eredita solo i compi con modificatori di visibilità come public e protected, mentre quelli private rimangono visibili solo nella classe base.

Il vantaggio dell'ereditarietà è il grande riutilizzo di codice già presente.

```
public class Persona {  
    private String name;  
    private String surname;  
  
    public Student (String n, String s){  
        name = n;  
        surname = s;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public String getSurname(){  
        return surname;  
    }  
}
```

```
public class Student extend Persona {  
    private int grade;  
    private int matricola;  
  
    public Student (String n, String s,  
                    int g, int m){  
        super (n,s);  
        grade = g;  
        matricola = m;  
    }  
  
    // metodi nuovi o  
    // sovrascritti
```

POLIMORFISMO

E' la possibilità di un'espressione di assumere valori diversi in base al tipo di dato su cui viene applicata.

Consente ad un oggetto di assumere più forme, infatti si può utilizzare un oggetto di una specifica classe come se fosse un oggetto della superclasse. Questo offre maggiore flessibilità nel progettare e scrivere codice.

Il polimorfismo viene applicato tramite:

overload = scrittura di più metodi con lo stesso nome, ma che hanno parametri in ingresso di tipo differente

override = riscrittura di un certo metodo ereditato da una superclasse

OVERLOADING E OVERRIDING:

Entrambi sono concetti legati al polimorfismo

Si parla di **overloading** (sovraccarico) quando in una stessa classe sono presenti più metodi con lo stesso nome, che eseguono operazioni diverse in base al tipo di input ricevuto, per questo hanno una lista di argomenti differenti grazie al quale si differenziano.

Si parla di **overriding** quando una sottoclasse sovrascrive alcuni metodi che ha ereditato dalla superclasse, mantenendo lo stesso nome e tipo di return (la stessa semantica), ma modificando il corpo della funzione, con un modificatore di visibilità non inferiore a quello della superclasse.

OVERLOAD

```
public class Calculator {  
    public int somma (int a, int b) {  
        return a+b;  
    }  
    public int somma (int a, int b, int c) {  
        return a+b+c;  
    }  
}
```

OVERRIDE

```
public class Square {  
    int base;  
    int height;  
    public void getInfo () {  
        System.out.println ("base: " + base + "\n");  
        System.out.println ("height: " + height);  
    }  
}  
  
public class ColoredSquare extends Square {  
    float color;  
    public void getInfo () {  
        super.getInfo ();  
        System.out.println ("color : " + color);  
    }  
}
```

INTERFACCIA

L'interfaccia è un tipo di dato **astratto** utilizzato per definire il comportamento di una o più classi.

Ha una struttura simile alle classi ma deve contenere metodi astratti e public, che poi devono essere obbligatoriamente sovrascritti da ogni classe che lo implementi.

Una classe implementa un interfaccia tramite la keyword "implements" e, a differenza dell'ereditarietà, una classe può implementare più interfacce, ma può avere una sola superclasse.

Un grande vantaggio è che i tipi di interfaccia rendono il codice maggiormente riutilizzabile, invece di modificare le classi.

```
public interface Measurable {  
    public double getMeasure()  
}
```

```
public class Rectangle implements Measurable {  
    private double base;  
    private double height;  
    public double getMeasure() {  
        return base * height;  
    }  
}
```

```
public class Circle implements Measurable {  
    private double radius;  
    public double getMeasure() {  
        return Math.PI * radius * radius;  
    }  
}
```

GESTIONE DELLE ECCEZIONI

In Java il sistema di gestione delle eccezioni è un modo per aggiungere quegli errori che possono verificarsi durante l'esecuzione di un programma. Usato per rendere più sicura la gestione di eventuali eccezioni, potendole cogliere e fornirne un modo per occuparsene senza interrompere il programma.

Esistono le **eccezioni controllate** dovute a circostanze esterne che il programmatore non può evitare, ma può prevedere. Ad esempio quando si gestiscono dati in entrata dove non si ha controllo su ciò che inserisce l'utente. Tutte le sottoclassi di IOException sono eccezioni controllate. È possibile gestire queste eccezioni con il try-catch, nella quale si inserisce il codice instabile che potrebbe generare errori nel corpo di codice dichiarato dalla keyword 'TRY' e poi si inserisce nel corpo con la keyword "CATCH" il tipo di errore che potrebbe essere lanciato e come dovrebbe essere gestito.

try {

```
Scanner console = new Scanner (System.in);
System.out.println ("Filename: ");
String filename = console.next();
FileReader reader = new FileReader (filename);
Scanner in = new Scanner (reader);
String input = in.next();
}
```

catch (FileNotFoundException e) {

```
System.out.println ("File non trovato");
}
```

Le **eccezioni non controllate** rappresentano un errore del programmatore e genera un'eccezione che non poteva essere prevista. Queste estendono la classe RuntimeException o Error ad esempio 'ArrayIndexOutOfBoundsException' e tanti altri.

ARRAYLIST VS ARRAY

L'ArrayList è molto simile all'array, ma tra loro ci sono alcune differenze:

- e' ArrayList è una classe che implementa l'interfaccia List ed è una struttura dinamica per cui puo' variare dimensione in base alle necessita', invece e' Array è una struttura statica, ovvero ha una grandezza fissata nel momento della creazione e non puo' essere modificata
- e' ArrayList contiene solo oggetti per questo i tipi primitivi hanno delle classi inviolate che possono essere usate con e' ArrayList
- un Array si presta meglio alla gestione della multidimensionalità, invece nell' ArrayList diventerebbe più complicato
- la classe ArrayList ha metodi che permettono di modificare la collezione in modo molto semplice. Ad esempio il metodo remove() permette con un solo parametro di rimuovere l'elemento e ridimensionare l'ArrayList. Invece, nell' Array la rimozione e il ridimensionamento sono operazioni lasciate al programmatore
- gli Array forniscono un controllo diretto sugli indici degli elementi, consentendo un accesso rapido e diretto a un elemento specifico attraverso la sua posizione nell' array, invece negli ArrayList non c'è un accesso diretto agli indici, ma viene richiesto l'utilizzo di metodi per accedere agli elementi

ARRAYLIST VS LINKEDLIST

- L' ArrayList è una classe che estende AbstractList e implementa l'interfaccia List, che utilizza una matrice dinamica per archiviare gli elementi della collezione. Invece, la LinkedList è una classe che estende AbstractSequentialList e implementa le interfacce List, Deque, Queue ed utilizza un elenco doppiamente collegato per memorizzare gli elementi dei dati

- L'accesso ai dati e' piu' lento nella LinkedList perche' ha un accesso sequenziale ai suoi dati, infatti ogni elemento contiene un riferimento al successivo. Invece l'ArrayList e' piu' efficiente avendo un accesso casuale agli elementi
- La manipolazione di elementi, invece, e' piu' lenta nell'ArrayList, ad esempio inserimento o rimozione richiedono il ridimensionamento dell'array e lo spostamento degli elementi, mentre nella LinkedList si devono aggiungere o rimuovere collegamenti all'interno della lista perciò e' piu' efficiente
- L'ArrayList richiede meno memoria rispetto alla LinkedList poiche' memorizza gli elementi in un array continuo, mentre l'altro per ogni elemento deve anche memorizzare un riferimento all'elemento successivo

ARRAYLIST VS SET

- L'ArrayList implementa l'interfaccia List che a sua volta appartiene all'interfaccia Collection. I Set, invece, sono un'interfaccia che fa parte dell'interfaccia Collection.
- Un ArrayList e' una collezione di oggetti memorizzati sequenzialmente, ammette duplicati ed e' dinamica. Per quanto riguarda l'accesso ai dati e' randomica (posizionale)
- I Set rappresentano valori senza duplicati, inoltre questa interfaccia e' implementata da HashSet e TreeSet: la prima utilizza una tabella hash, quindi e' disordinata ma offre in tempo costante operazioni come la ricerca, la rimozione e l'inserimento. Invece, TreeSet utilizza un albero binario di ricerca ed e' ordinato, ed e' piu' lento per le operazioni di inserimento, rimozione o ricerca, ma ha un accesso diretto agli elementi

HASHSET E TREESSET

corrispondono a un set, insieme di valori di tipo Object non doppioni e differiscono per l'implementazione interna

HashSet corrisponde
una tabella hash
NON ORDINATO

TreeSet corrisponde
un albero binario di ricerca
ORDINATO

HASHMAP e TREEMAP

entrambe corrispondono ad un dizionario in grado di associare una chiave di tipo Object ad un valore di tipo Object

↓

differiscono per un IMPLEMENTAZIONE INTERNA

Hashmap corrisponde
una tabella hash
NON ORDINATA

TreeMap corrisponde
un albero binario di ricerca
ORDINATO

ARRAYLIST VS HASHSET

- L'ArrayList implementa l'interfaccia List, invece HashSet implementa l'interfaccia Set
- L'ArrayList ricrea un array, mentre HashSet utilizza le tabelle hash per la sua implementazione interna
- L'ArrayList mantiene l'ordine in cui gli elementi sono inseriti mentre l'HashSet è una collezione disordinata.
- Possono esserci duplicati nell'ArrayList, mentre nel HashSet no

SOLID

S - single responsibility principle

ogni classe deve avere un'unica responsabilità, encapsulata in sé

↳ ogni compito è assegnato ad un'istanza separata

O - open/closed principle

Un software dovrebbe essere aperto alle estensioni e chiuso alle modifiche

↳ non deve avere attributi troppo specifici da impedire l'estensione

L - liskov substitution principle

Gli oggetti devono poter essere sostituiti con dei loro sottotipi

↳ se una classe implementa la classe Animal, non ci devono essere modifiche al programma se lo sostituisco con la classe Cone

I - interface segregation principle

Prefabbricati più interfacce specifiche che una generica

↳ Measurable è troppo generico

D - dependency inversion principle

Una classe dovrebbe dipendere dalle astrazioni, non da classi concrete

↳ un codice implementante di una classe e le sue sottoclassi deve essere scritto basandosi sulla superclasse, non sulle sottoclassi

