SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA
FACOLTÀ DI INFORMATICA

Metodologie di Programmazione

Definizioni teoriche e documentazione rapida

Author
Simone Bianco

5 marzo 2023

Indice

0 Programmazione Orientata agli Oggetti **1**
- 0.1 Approccio Object Oriented . . . 1
- 0.2 Classi ed Oggetti . . . 2
  - 0.2.1 Ereditarietà . . . 3
  - 0.2.2 Polimorfismo . . . 3
  - 0.2.3 Modificatori di Visibilità . . . 4
  - 0.2.4 Modificatori Static e Final . . . 5
- 0.3 Convenzioni nelle Classi . . . 5
  - 0.3.1 Coesione . . . 5
  - 0.3.2 Consistenza . . . 5
  - 0.3.3 Metodi Accessors e Mutators . . . 5
  - 0.3.4 Classi Attori e Starter . . . 6
- 0.4 Interfacce . . . 6

1 Principi SOLID **8**

2 Lettura e Scrittura da File **9**
- 2.1 Lettura tramite classe Scanner . . . 9
- 2.2 Scrittura tramite classe PrintWriter . . . 10
- 2.3 Metodi utili classe Scanner e PrintWriter . . . 10

3 Strutture dati utili **11**
- 3.1 Classi Map . . . 11
  - 3.1.1 HashMap e TreeMap . . . 11
- 3.2 Classi Set . . . 12
  - 3.2.1 HashSet e TreeSet . . . 12

Capitolo 0

Programmazione Orientata agli Oggetti

0.1 Approccio Object Oriented

La **Programmazione Orientata agli Oggetti**, oppure **object-oriented programming (OOP)** è un modo sistematico di scrivere codice (paradigma di programmazione), secondo cui vengono definite delle entità software chiamate **oggetti** in grado di interagire gli uni con gli altri attraverso lo scambio di **messaggi**.

Tale paradigma risulta particolarmente adatto nei contesti in cui si possono definire delle **relazioni di interdipendenza** tra i concetti da modellare:

- Un oggetto di tipo `Libro` è composto da molti oggetti di tipo `Capitolo`, a loro volta composti da molti oggetti di tipo `Pagina` (**composizione**)
- Un oggetto di tipo `Automobile` è uno specifico tipo di `Veicolo` (**specializzazione**)
- Un oggetto di tipo `LettoreDVD` necessita un oggetto di tipo `DVD` (**utilizzo**)

Per via della sua natura concettuale la programmazione OOP risulta in grado di:

- Fornire un **supporto naturale** alla modellazione software degli oggetti del mondo reale o del modello astratto da riprodurre
- Permettere una più facile gestione e manutenzione di **progetti di grandi dimensioni**
- Rendere il codice **modulare** e **riusabile**

0.2 Classi ed Oggetti

Una **classe** è un tipo di dato astratto rappresentante un **elemento** costituito secondo determinate **caratteristiche** (attributi) e **operazioni** (metodi) eseguibili su di esso.

Immaginiamo star definendo una classe `Persona` secondo le caratteristiche ed operazioni che la definiscono.

- Le sue caratteristiche comprenderanno sicuramente:
  - Una stringa contenente un *nome*
  - Una stringa contenente un *cognome*
  - Un intero contenente un'*altezza* in centimetri
  - Una intero contenente un *peso* in chilogrammi
- Le operazioni che può svolgere comprendono *mangiare*, *bere*, *dormire*

Dunque, possiamo modellare la classe `Persona` nel seguente modo:

Person
name : String surname : String height : int weight : int
eat() : void drink() : void sleep() : void

In Java, tale classe viene implementata come:

```
public class Person{

    private String name, surname;
    private int height, weight;

    public Person(String name, String surname, int height, int weight){
        this.name = name;
        this.surname = surname;
        this.height = height;
        this.weight = weight;
    }

    public void eat(){...}

    public void drink(){...}

    public void sleep(){...}
}
```

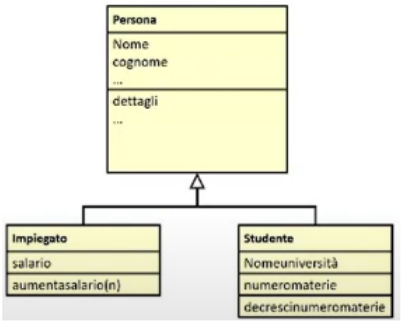
Una volta definita la classe `Persona`, essa può essere utilizzata per **istanziare** (ossia allocare spazio in memoria) un **oggetto** di tipo Persona.

0.2.1 Ereditarietà

L'**ereditarietà** è una tecnica della programmazione ad oggeti tramite cui è possibile creare una **sottoclasse** (o classe derivata) in gradi di **estendere** le caratteristiche (attributi) e funzionalità (metodi) della **superclasse** (o classe base) da cui esser derivano.

Una sottoclasse, **eredita** tutti gli attributi e metodi della superclasse, potendo **estenderli** con dei propri oppure **ridefinendo** il funzionamento dei metodi ereditati (*Override*).

Dunque, una superclasse modella un **concetto generico**, mentre una sottoclasse modella un **concetto più specifico**, potendo accedere ai metodi della superclasse tramite la sintassi *super.method()*



Una sottoclasse può **ridefinire i metodi** della superclasse a patto che mantenga:

- Lo stesso nome e parametri
- Lo stesso tipo di valore di return
- La stessa semantica

Tale processo viene definito col termine ***Overriding*** e viene utilizzato per rendere la sottoclasse più specifica.

0.2.2 Polimorfismo

Il vantaggio maggiore all'interno della OOP viene fornito dal **polimorfismo**, ossia la possibilità di un'espressione di assumere valori diversi in base al tipo di dato su cui viene applicata. In particolare, il polimorfismo viene applicato tramite:

- **Overriding**: sovrascrittura di metodi ereditati da una superclasse.
- *Esempio*:

```
public class Dog extends Animal{
    @Override
    public class eat(){
        ...  // sovrascrittura del metodo
    }
}
```

- **Overloading**: implementazione di più metodi con lo stesso nome ma con argomenti e valori di return diversi all'interno della stessa classe, eseguendo operazioni diverse a seconda degli input ricevuti.

  Esempio:

```
public void setDate(int day, int month, int year){
    this.date = new Date(day, month, year);
}

public void setDate(String date){
    String[] parts = date.split("/");

    day = Integer.parseInt(parts[0]);
    month = Integer.parseInt(parts[1]);
    year = Integer.parseInt(parts[2]);

    this.setDate(day, month, year);
    // utilizzo dell'altro metodo definito (riuso del codice)
}
```

0.2.3 Modificatori di Visibilità

I modificatori di visibilità di accesso determina la possibilità di accedere ad attributi e metodi di una classe da parte di altre classi.

I modificatori possono essere implementati su due livelli:

- A **livello di classe**, venendo applicato sull'intera classe
- A **livello di attributo o metodo** all'interno di una classe, venendo applicato solo su di esso

Esistono quattro tipi di **modificatori di visibilità**:

- **`public`**: visibile da qualsiasi classe
- **`private`**: visibile solo all'interno della classe stessa
- **`protected`**: visibile nella classe stessa, nelle sue sottoclassi e dalle classi appartenenti allo stesso pacchetto
- ***Nessun modificatore***: visibile nella classe stessa e dalle classi appartenenti allo stesso pacchetto

Modificatore	Classe	Pacchetto	Sottoclassi	Tutti
`public`	Sì	Sì	Sì	Sì
`protected`	Sì	Sì	Sì	
no-modifier	Sì	Sì		
`private`	Sì			

0.2.4 Modificatori Static e Final

Oltre ai modificatori di visibilità, sono presenti due ulteriori modificatori, i quali assumono un comportamento diverso a seconda dell'elemento a cui vengono associati:

- `final`:
  - Se usato su una **classe**, essa non potrà avere sottoclassi
  - Se usato sul **metodo** di una classe, le sue sottoclassi non potranno effettuare un override di quel metodo
  - Se usato sull'**attributo** di una classe, esso non potrà essere modificato
- `static`:
  - Se usato sul **metodo** di una classe, esso sarà accessibile senza dover istanziare un oggetto di tale classe
  - Se usato su un **attributo**, esso sarà condiviso tra tutte le istanze di tale classe

0.3 Convenzioni nelle Classi

0.3.1 Coesione

- Una classe deve rappresentare un **singolo concetto** e tutte le azioni collegate ad esso. Se mescolo più livelli concettuali nella stessa classe, sto rompendo l'idea della **coesione**, è necessario quindi creare classi separate che si occupano di uno specifico concetto.

0.3.2 Consistenza

- I nomi delle variabili e classi devono adeguarsi al lavoro svolto, attenendosi alle **Java Code Convention**:
  - I **nomi delle classi** devono essere scritti in PascalCase.

    Esempio: `Person` e `Animal`
  - Gli **attributi** devono essere scritti in camelCase.

    Esempio: `longVariableName` e `getVariable()`
  - Gli **attributi costanti** devono essere scritti in maiuscolo.

    Esempio: `MAX_HEIGHT`

0.3.3 Metodi Accessors e Mutators

- Un metodo **accessor** è un metodo che chiede all'oggetto di svolgere operazioni (ed eventualmente restituire un risultato) senza modificare lo stato interno dell'oggetto.

  Esempio: una classe `Shape` con dei metodi `computeArea()` o `getSides()`
- Un metodo **mutator** è un metodo che modifica lo stato dell'oggetto, ritornando sempre `void`

Esempio: solitamente viene implementato come `public void setVariable()`

- Una **classe immutabile** è una classe senza alcun metodo mutator

  Esempio: la classe `String` nativa di Java

0.3.4 Classi Attori e Starter

- **Classi attori**: svolgono un'azione e solitamente finiscono con *"-er"* o *"-or"*
- **Classi starter**:classi che contengono il main

0.4 Interfacce

Le **interfacce** sono un meccanismo di completa astrazione in grado di **descrivere il comportamento di altre classi**.

Le interfacce risultano essere molto diverse dalle classi:

- Esse possono essere **implementate** da una classe tramite la keyword `implements` (e non `extends`)
- Una classe può implementare un **numero indeterminato** di interfacce a differenza dell'estensione di una singola superclasse
- **Non hanno metodi costruttori** e non possono essere costruiti oggetti di tipo interfaccia
- Definiscono gli attributi e i metodi che devono **obbligatoriamente** essere **sovrascritti** nelle classi che le implementano
- Tutti i **metodi** di un interfaccia sono **pubblici** ed **astratti**, ossia non hanno una vera implementazione al loro interno

```
public interface Measurable{

    double getMeasure();
    // ogni classe implementante deve
    // sovrascrivere questo metodo
}

public class Rectangle implements Measurable{

    ... // attributi di Rectangle

    @Override
    public double getMeasure(){
        ...
    }
}
```

Dopo aver dichiarato che la classe `Rectangle` implementa l'interfaccia `Measurable`, gli oggetti di tipo `Rectangle` **sono anche di tipo** `Measurable`:

```
Measurable obj = new Rectangle();
```

Una variabile di tipo `Measurable` può contenere un riferimento a un oggetto che sia esemplare di una **classe qualsiasi che implementa l'interfaccia** `Measurable`.

Un **metodo di default** in un'interfaccia è un metodo **non statico** di cui viene definita anche un'**implementazione predefinita** di tale metodo, la quale viene ereditata dalle classi implementanti, senza necessità che esse sovrascrivano tale metodo.

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

    default boolean smallerThan(Measurable other){
        boolean isSmaller = getMeasure() < other.getMeasure();
        return isSmaller;
    }
}
```

Durante l'uso delle interfacce si potrebbero verificare dei **conflitti**:

- Se una sottoclasse **estende** una superclasse ed **implementa** un interfaccia, esse potrebbero possedere un metodo definito con lo stesso nome, stesso return e stessi argomenti in input.

  In tal caso, la sottoclasse erediterà il metodo della superclasse (**ereditarietà vince su implementazione**)

- Se una classe implementa **due interfacce** che hanno un metodo di default definito con lo stesso nome, stesso return e stessi argomenti in input, allora la sua **sovrascrittura** all'interno della classe implementante sarà **obbligatoria**

Capitolo 1

Principi SOLID

S - Single responsibility principle

- Ogni classe dovrebbe avere una ed una sola responsabilità, interamente **incapsulata** al suo interno.
- ***Esempio***: se esiste una classe `Database` in grado di creare una connessione, leggere e modificare i dati del database, ogni compito deve essere affidato ad un'istanza separata (`ConnectionHandler`, `DataReader`, `DataWriter`)

O - Open/closed principle

- Un'entità software dovrebbe essere aperta alle **estensioni**, ma chiusa alle modifiche.
- ***Esempio***: una classe non dovrebbe possedere attributi **troppo specifici**, poiché impedirebbe la possibilità di estenderne il funzionamento.

L - Liskov substitution principle

- Gli oggetti dovrebbero poter essere sostituiti con dei loro **sottotipi**, senza alterare il comportamento del programma che li utilizza.
- ***Esempio***: se una classe implementa un'istanza della classe `Animale`, sostituirla con la classe `Cane` non deve comportare modifiche al programma.

I - Interface segregation principle

- Sarebbero preferibili **più interfacce specifiche**, che una singola generica.
- ***Esempio***: un'interfaccia `Measurable` risulta troppo generica, poiché ogni oggetto è potenzialmente misurabile in qualche modo.

D - Dependency inversion principle

- Una classe dovrebbe dipendere dalle **astrazioni**, non da classi concrete.
- ***Esempio***: se esiste una classe `Animale` e varie sue sottoclassi (`Cane`, `Gatto`, ...), il codice implementante un'istanza di tali classi deve essere scritto basandosi sulla classe `Animale` e non sulle sue sottoclassi.

Capitolo 2

Lettura e Scrittura da File

2.1 Lettura tramite classe Scanner

```
public ArrayList<String> readFile(String path){

    ArrayList<String> lines = new ArrayList<String>();
    Scanner scanner;
    String line;
    File file;

    try{
        file = new File(path);
        scanner = new Scanner(file);

        while(scanner.hasNextLine()){

            line = scanner.nextLine();

            lines.add(line);
        }
    }
    catch(FileNotFoundException e){
        // gestione dell'eccezione
    }
    finally{
        scanner.close();
    }

    return lines;
}
```

2.2 Scrittura tramite classe PrintWriter

```
public void writeFile(String path, ArrayList<Strings> lines){

    PrintWriter writer;

    try{
        writer = new PrintWriter(path);

        for(String line : lines){
            writer.println(line);

            //oppure

            writer.print(line);
            writer.print("\n");
        }
    }
    catch(FileNotFoundException e){
        // gestione dell'eccezione
    }
    finally{
        writer.close();
    }
}
```

2.3 Metodi utili classe Scanner e PrintWriter

Scanner

- `hasNext()`: restituisce True se c'è una prossima parola leggibile
- `hasNextLine()`; restituisce True se c'è una prossima riga leggibile
- `next()`: legge la prossima parola (usa lo spazio come delimitatore)
- `nextLine()`: legge la prossima riga (usa
  n come delimitatore)

PrintWriter

La classe `PrintWriter` possiede letteralmente gli stessi identici metodi di `System.out`

Capitolo 3

Strutture dati utili

3.1 Classi Map

3.1.1 HashMap e TreeMap

Entrambe le classi corrispondono ad un dizionario in grado di associare una **chiave univoca** di tipo `Object` ad un **valore** di tipo `Object`

Esse differiscono solo per metodo di **implementazione interna**, poiché un `HashMap` corrisponde ad una tabella hash (di conseguenza non ordinata), mentre un `TreeMap` corrisponde ad un albero binario di ricerca (di conseguenza ordinato).

- `containsKey(Object key)`: restituisce True se la chiave è associata ad un valore nel dizionario
- `containsValue(Object value)`: restituisce True se il valore è associato ad una chiave nel dizionario
- `put(Object key, Object value)`: associa nel dizionario la chiave al valore in input
- `get(Object key`: restituisce il valore associato alla chiave data in input (se presente nel dizionario)
- `remove(Object key)`: rimuove dal dizionario l'associazione definita secondo la chiave data in input (se presente nel dizionario)
- `size()`: restituisce la quantità di oggetti internamente associati

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

    HashMap<String, int> hashmap = new HashMap<String, int>();

    TreeMap<String, int> treemap = new TreeMap<String, int>();

    //String e int possono essere sostituiti da qualsiasi tipo
    //poiché le classi fanno uso di Generics
}
```

3.2 Classi Set

3.2.1 HashSet e TreeSet

Entrambe le classi corrispondono ad un set, ossia un insieme di valori di tipo `Object` **non doppioni** (ossia non identici tra di loro).

Esse differiscono solo per metodo di **implementazione interna**, poiché un `HashSet` corrisponde ad una tabella hash (di conseguenza non ordinata), mentre un `TreeSet` corrisponde ad un albero binario di ricerca (di conseguenza ordinato).

- `contains(Object value)`: restituisce True se il valore è presente nel set
- `add(Object value)`: aggiunge il valore al set
- `remove(Object value)`: rimuove il valore dal set
- `size()`: restituisce la quantità di elementi presenti

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

    HashSet<String> hashset = new HashSet<String>();

    TreeSet<String> treeset = new TreeSet<String>();

    //String può essere sostituito da qualsiasi tipo
    //poiché le classi fanno uso di Generics
}
```