Università degli Studi di Verona

CORSO DI LAUREA IN INFORMATICA

Programmazione II

Federico Brutti
federico.brutti@studenti.univr.it

Inserire citazione inerente alla materia

Contents

Chapter 1

Introduzione

1.1 Programmazione Orientata agli Oggetti

Per **Programmazione orientata agli oggetti**, o OOP, si intende un particolare paradigma di programmazione strutturato intorno agli oggetti; delle strutture di dati contenenti attributi e metodi. Il software scritto in un linguaggio orientato agli oggetti baserà il suo funzionamento secondo le interazioni fra questi elementi.

Il vantaggio principale di questo modus operandi è nel rendere il codice estremamente modulare, riutilizzabile e mantenibile, in quanto diviso in sezioni precise. I due elementi base della OOP sono le **classi** e gli **oggetti**. Le prime definiscono una struttura dati, alla quale è possibile associare dati, detti **attributi**, e funzioni, chiamate **metodi**. Dalle classi, che in parole povere fungono da tipo di dato, è possibile ottenere gli oggetti, delle loro istanze, con stessi dati e metodi. A partire da questi elementi, definiamo i principi della programmazione orientata agli oggetti:

- **Incapsulamento**: Restrizione dell'accesso ai dati di un oggetto.
- **Astrazione**: Nascondere dettagli di implementazione e mostrare solamente le caratteristiche essenziali dell'oggetto.
- **Ereditarietà**: Tramandare campi e metodi ad altri elementi a partire da una superclasse.
- **Polimorfismo**: Possibilità di modificare campi e metodi delle singole istanze.

Un buon iter di lavoro generale per lavorare con un linguaggio orientato a oggetti è dato da **identificare** classi e oggetti necessari, per poi **dichiararle** insieme a relativi metodi e attributi utili. Dopodiché bisognerà **definire** le modalità di interazione fra gli oggetti ed infine **minimizzarne** quante più possibile.

Essendo che si andrà a lavorare su progetti di medie o grandi dimensioni, risulta essere buona prassi anche una corretta documentazione del codice tramite **Unified Model**

Languages, i quali consentono di creare una descrizione grafica di classi e oggetti. Verranno approfonditi più avanti nel corso.

1.2 Funzionamento di Java

Il linguaggio utilizzato nel corso sarà **Java**, nato nel '95 e proprietà di Oracle, è diventato lo standard nel mercato del lavoro per il software development. La sua caratteristica principale è la portabilità, ovvero è possibile eseguire programmi su qualunque architettura grazie alla **Java Virtual Machine**, la quale funge da interprete al codice compilato.

Più precisamente, dopo la compilazione, verrà creato in output un file con estensione ".class" contenente il **bytecode**. Questo codice è ciò che viene effettivamente interpretato in runtime da una parte della JVM, il compiler **Just In Time**, utile anche per ulteriori ottimizzazioni. In soldoni, a patto che la macchina abbia installata la JVM, sarà possibile eseguire i files compilati. Altre caratteristiche di Java sono:

- Linguaggio fortemente tipizzato, ovvero è possibile dichiarare variabili di un determinato tipo di dato, le quali non lo possono cambiare una volta aggiunte al codice.
- Non ha manipolazioni esplicite di puntatori grazie alla filosofia dell'incapsulamento, rendendo meno probabili errori riguardanti la memoria.
- Controlla il runtime, rendendo impossibile avere array overflow.
- Il **Garbage collector** domina la memoria dinamica, alloca e dealloca dove necessario. Inoltre gestisce memory leak.
- È possibile usare eccezioni per controllare gli errori.
- Linguaggio fortemente dinamico, poiché fa loading e linking in runtime. Inoltre, usa dimensioni di array dinamiche.

Dove è possibile installare sulla macchina solo la JVM, per scrivere programmi in Java è necessario usare il **Java Development Kit**, compreso di debugger, compiler, disassembler, ed un applicativo per la documentazione.

Per l'esecuzione dei programmi abbiamo poi il **Java Runtime Environment**, avente con sé librerie di classe, il compiler JIT precedentemente menzionato, la JVM e il Java application launcher.

1.3 Struttura, compilazione ed esecuzione dei programmi

Anzitutto, i programmi scritti in Java hanno estensione ".java" e vedono i blocchi di codice completamente all'interno di una classe, il cui nome deve essere uguale a quello del file. Al suo interno è poi necessario dichiarare l'entry point tramite il metodo **main**.

```
1  // Per compilare: javac file1.java file2.java ... filen.java
2  // Per eseguire: java file1 file2 ... filen
3
4  // Dichiarazione di classe, il nome coincide con quello del file.
5  public class HelloWorld {
6    // Entry point del programma, metodo di nome main
7    public static void main (string[] args) {
8      // Blocco di codice
9    }
10 }
```

Post-compilazione, avremo in output un file di estensione ".class". Infatti, ogni file è visto come classe il cui caricamento in compilazione è basato sul **classpath**, la lista di locazioni dove le classi possono essere prese. Se il compiler non trova una classe, lancerà un'eccezione e non verrà creato l'eseguibile.

Questo procedimento è valido se si lavora da terminale. È consigliato l'utilizzo di un IDE per la semplificazione del lavoro; nel corso verrà usato IntelliJ, il quale consente anche di automatizzare il deployment tramite **build tools** come Maven, che vedremo nella prossima sezione.

1.4 Gestione di un progetto su più file

Perché limitarsi ad un singolo file quando è possibile dividere in **unità** ogni funzione? Abbiamo visto nello snippet precedente che possiamo compilare ed eseguire più classi allo stesso tempo, ma la scrittura è estremamente tediosa, inefficiente e soggetta ad errori. La soluzione si ha in due passaggi:

- Una corretta divisione in cartelle del progetto.
- Raggruppamento delle classi in un unico pacchetto.

Il primo passo riguarda uno studio per ingegneria del software, quindi concentriamoci sulla seconda parte. Non è buona prassi, ma generalmente, quando in un file .java non è indicata l'appartenenza ad un pacchetto, il compilatore lo assocerà a quello di default, il quale non ha un nome e non consente ad altre classi di accedervi. Quindi è consigliato specificare l'appartenenza ad un dato pacchetto prima della definizione di classe; ciò consentirà di avere un classpath più compatto e più facile da gestire.

Se volessimo poi gestire facilmente più pacchetti, avremo bisogno di un contenitore più astratto: un file ".jar". Si tratta di un archivio compresso di bytecode e altre metainformazioni; idealmente, si ha un jar per progetto.

```
// Compila: javac -d bin/ src/pkg/MyClass.java
// Impacchetta: jar cvf MyJar.jar -C bin/ pkg/
// Esegui: java -cp MyJar.jar pkg.MyClass

// Segnala che la classe MyClass sta all'interno del pacchetto pkg.
package pkg;

public class MyClass {
  public static void main(String[] args) {
    // Blocco di codice
  }
}
```

Bisogna tuttavia specificare quale sia la classe main; a questo scopo vengono in aiuto le metainformazioni menzionate prima. Tramite un file speciale chiamato "manifest.txt" è possibile fare non solo questo, ma anche specificare quali classi compilare.

```
// Impacchetta: jar cvfm MyJar.jar manifest.txt -C bin/ pkg/
// Esegui: java -jar MyJar.jar

// manifest.txt
Main-Class: pkg.MyClass
```

Chapter 2

Il linguaggio Java

2.1 Tipi primitivi, variabili e costanti, operatori e assegnamenti

Supponendo che tu abbia frequentato o quantomeno ascoltato il corso di programmazione 1, la lettura risulterà nettamente più scorrevole. Infatti Java, per quanto riguarda le informazioni elementari, condivide una sintassi quasi uguale a quella di C, composta da:

- **Parole chiave del linguaggio**: Hanno un significato speciale e non possono essere usate per la dichiarazione di variabili o funzioni.
- **Identificatori**: I nomi scelti per gli elementi di programmazione definiti nel linguaggio.
- **Operatori**: Simboli per effettuare operazioni.
- **Dati**: Valori delle variabili, le informazioni nel codice.

Si mantengono tutte le funzioni legate al linguaggio C per quanto riguarda tipi primitivi, operatori ed assegnamenti. Abbiamo quindi: **int**, **double**, **char**, aggiungendo **boolean** e **String**, quest'ultimo non primitivo, ma successivamente approfondito.

Di base ogni dato è considerato una variabile se non specificato altrimenti. Infatti, per la dichiarazione di costanti sarà necessario aggiungere la keyword **final**.

2.2 Definizione di classe, this., parametri formali e attuali

2.3 Metodo costruttore, costruttore di default e overloading

2.4 Dichiarazione di classi e oggetti

Parliamo adesso di codice effettivo. Per prima cosa bisogna dichiarare una classe con i relativi campi. Quindi:

```
1  // Dichiarazione della classe Date
2  public class Date {
3
4    // Scope dei campi della classe, questi sono attributi...
5    int day, month, year;
6
7    // ...e questo un metodo.
8    String printDate() {
9      return day + "/" + month + "/" + year;
10   }
11 }
```

Si può estendere il discorso introducendo i concetti di **modificatore** e **funzione costruttore**. Il primo indica la modalità di accesso alla struttura e può essere usato sia per classi che oggetti, ed il secondo è un metodo dello stesso nome della classe che viene eseguito alla creazione di un nuovo oggetto. Per esempio, consideriamo il seguente codice all'interno della classe Date, dichiarata prima:

```
1  // Funzione costruttore. Qui popola le variabili dell'oggetto.
   Modificatore public.
2  public Date(int day, int month, int year) {
3    // "this." indica la locazione dell'oggetto corrente.
4    this.day = day;
5    this.month = month;
6    this.year = year;
7  }
8
9  // Metodo isFirstDay con modificatore static. Non ha accesso a campi
   non statici.
10 static boolean isFirstDay(int day) {
11   if (day == 1) return true;
12   else return false;
13 }
```

Nello specifico, per quanto riguarda i modificatori, abbiamo **public**, rendendo accessibili i dati senza limitazioni e **private**, che blocca l'accesso dall'esterno della classe. Questi specificano le modalità di accesso. Un altro modificatore presente nello snippet è **static**, che rende il metodo utilizzabile a prescindere dall'oggetto. Vediamo ora come istanziare un oggetto, a partire dagli snippet del file Date.java di prima:

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

    // Utilizzo di isFirstDay fuori oggetto. Possibile dato static.
    System.out.println(Date.isFirstDay(1));

    // Creazione dell'oggetto today e stampa con il metodo printDate.
    Date today = new Date(14, 10, 2024);
    System.out.println(today.printDate());
  }
}
```

Come hai potuto vedere, l'istanziazione avviene con la keyword **new**, la quale alloca spazio sufficiente per contenere i dati dell'oggetto, mentre per accedere ai singoli campi si utilizza l'operatore **dot**, ovvero '.'.

Con queste nozioni si hanno strumenti a sufficienza per scrivere dei programmi base in Java. Con l'aggiunta della sezione successiva si potrà iniziare a strutturare dei progetti.

2.5 Incapsulamento e information hiding

Per **incapsulamento** si intende un raggruppamento di dati e metodi, i quali lavorano sui primi. Consente di avere una visuale più compatta riguardo ai campi di ogni classe. Se alcuni campi sono poi dichiarati con il modificatore **private**, l'incapsulamento esprime il suo massimo potenziale di sicurezza, consentendo l'**information hiding** e rendendo accessibili i campi solamente tramite gli appositi metodi.

```
public class Person {
  // Dichiarare senza modificatori setta public di base
  String name;
  private int age = -1;

  Person(String name) { this.name = name; }

  // Metodo per segnalare se age risulta illegale
  void changeAge(int age) {
    if (age > 0 && age <= 122) this.age = age;
    else System.out.println("Invalid age.");
  }
}
```

La variabile age in questo caso diventa accessibile solo tramite il metodo changeAge, perché risulta inaccessibile al di fuori. La corretta gestione di questa dinamica si ha introducendo due metodi appositi, il **getter**, che riceve il dato per poi assegnarlo all'oggetto, ed il **setter**, che si assicura sia in un formato corretto. Nello snippet appena visto, changeAge è inserito come setter.

2.6 Array, matrici e tipo enum

In Java è possibile dichiarare **array** mono- e bidimensionali; sono visti come oggetti speciali allocati in heap, quindi definiti in runtime, e da un punto di vista pratico sono sequenze di puntatori ad oggetto. Alcuni aspetti importanti sono:

- In mancanza di inizializzazione, la JVM li setta a null.
- Post-dichiarazione, sarà impossibile modificarne la lunghezza.
- Essendo sequenze di puntatori, il metodo equals non funzionerà come negli oggetti normali.

```
// Dichiarazione di array
int[] arr = new int[dimensione];
// Scrittura in un indice specifico
arr[numeroIndice] = NomeClasse(eventuali_parametri);
// Accesso ad un elemento in un indice specifico
int var = arr[numeroIndice];
```

Per una manipolazione corretta degli array ci corre in aiuto **java.util.Arrays**, contenente metodi utili per fare confronti e stampare il contenuto. Questi sinergizzano ovviamente col ciclo for, il quale ha una forma enhanced con una sintatti più compatta.

```
// Enhanced for loop, equivalente alla scrittura classica
for (tipoVariabile : espressioneIterabile);
// Confronta se due array hanno gli stessi elementi
Arrays.equals(arr1, arr2);
// Stampa ogni elemento dell'array in formato stringa
Arrays.toString(arr);
```

Ovviamente, nella creazione di matrici, è possibile usare tutti i metodi appena visti; ci sono poche differenze rispetto a quanto già visto in C.

```
// Dichiarazione ed inizializzazione di una matrice di interi
int[][] matrix = {{1,2,3}, {4,5,6}, {7,8,9}};
// Per scambiare facilmente righe o colonne puoi prendere il
 puntatore.
int[] tmp = matrix[0];
matrix[0] = matrix[matrix.length-1];
matrix[matrix.length-1] = tmp;
```

```
7  // Stampa gli elementi di una matrice
8  for (int[] row:matrix) System.out.println(Arrays.toString(row));
```

Infine, abbiamo, sempre come in C, i tipi per le **enumerazioni**. Rappresentano un insieme finito di costanti e risultano utili per definire valori arbitrari utilizzati più volte, non richiedendo la dichiarazione di variabili per ogni blocco di codice. Vengono anche loro con alcuni metodi utili:

```
1  // Dichiarazione
2  public enum nomeEnum {
3    CST1, CST2, ...
4  }
5
6  // Ritorna il nome della costante assegnata all'oggetto
7  nomeObjEnum.name(nomeOggetto.campo);
8  // Ritorna il valore della costante assegnata all'oggetto
9  nomeObjEnum.ordinal(nomeOggetto.campo);
```

Chapter 3

Librerie utili

3.1 Libreria digitale

Le librerie digitali di Java sono un insieme di funzioni contenute in pacchetti appositi e fornite da terze parti. Come altri utenti ne hanno scritte, così potremmo fare anche noi. In ogni caso, tutti i file da eseguire sono visti come classi, ed infatti dovranno essere compresi nel classpath, se non sono standard.

Per utilizzare una libreria è necessario importarla nel file desiderato con la keyword **import**. Per esempio, nella libreria java.util è presente la classe Scanner, che viene usata per ricevere input da tastiera.

```
1  // Aggiunge il pacchetto Scanner dal path java/util/Scanner
2  import java.util.Scanner;
3
4  public class Mult {
5    public static void main(String args[]) {
6
7      // Dichiarazione dell'oggetto keyScan di classe Scanner
8      Scanner keyScan = new Scanner(System.in);
9      int n1, n2;
10
11     System.out.print("Inserisci il primo fattore: ");
12     // Assegna a n1 l'intero letto da tastiera, idem n2.
13     n1 = keyScan.nextInt();
14     System.out.print("Inserisci il secondo fattore: ");
15     n2 = keyScan.nextInt();
16
17     // Chiudi lo scanner con il metodo close().
18     keyScan.close();
19     System.out.println("Risultato: " + n1*n2);
20   }
21 }
```

3.2 java.lang.*

Il pacchetto java.lang.* comprende la libreria nativa del linguaggio, contenente tutte le funzioni di base come costrutti, oggetti e altri strumenti.

3.2.1 .System

3.2.2 .String, StringBuilder

3.2.3 .Math, .Random

3.2.4 .Object

3.2.5 .Integer, .Character

3.2.6 .AutoCloseable

3.2.7 .Iterable

3.2.8 .Thread

3.3 java.util.*

Il pacchetto java.util.* contiene varie funzioni di utility per rendere il codice più leggibile, astratto, modulare e sicuro.

3.3.1 .Scanner

3.3.2 .Arrays

3.3.3 .Iterator

3.3.4 Strutture dati

java.util.Collection¡E¿, java.util.List¡E¿, java.util.Queue¡E¿, java.util.Set¡E¿, java.util.LinkedList¡E¿, java.util.ArrayList¡E¿, java.util.PriorityQueue¡E¿ java.util.HashSet¡E¿, java.util.SortedSet¡E¿, java.util.TreeSet¡E¿, java.util.Map¡K,V¿, java.util.HashMap¡K,V¿, java.util.SortedMap¡K,V¿, java.util.TreeMap¡K,V¿

3.4 java.io.*

Il pacchetto java.io.* fornisce funzioni relative alla gestione di un flusso di dati, sia in input, che in output.

3.4.1 .Closeable

3.4.2 .Reader, .FileReader, .BufferedReader

3.4.3 .Writer, .FileWriter, .BufferedWriter, .PrintWriter

Chapter 4

Boh, da vedersi