Università degli Studi di Verona

CORSO DI LAUREA IN INFORMATICA

Programmazione Java

Federico Brutti
federico.brutti@studenti.univr.it

Inserire citazione inerente alla materia

Contents

Chapter 1

Introduzione

1.1 Programmazione orientata agli oggetti

Il corso si propone di fornire le conoscenze necessarie alla comprensione, sviluppo e correzione di software realizzato in un linguaggio di programmazione orientato agli oggetti e di fornire le competenze relative alla strutturazione di progetti software di medie dimensioni.

Lo scopo e l'utilità della OOP è quello di gestire appropriatamente un programma con molte linee di codice. Rende il codice estremamente modulare, riutilizzabile e di conseguenza risulta più facile mantenerlo. Per il modo in cui sono trattati gli oggetti, inoltre, il codice è reso particolarmente sicuro.

Familiarizziamo ora con i due concetti base della OOP: **classi** e **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. Infatti, il processo di creazione di un oggetto è detto **istanziazione**.

Il workflow della OOP si basa sulle interazioni fra gli oggetti; queste avvengono grazie alle loro **interfacce**, set di messaggi che l'oggetto può ricevere, mappate ad un metodo nello stesso. Se si riceve un messaggio al di fuori dell'interfaccia, è detto illegale e viene bloccato. Un iter di lavoro generale per una corretta programmazione a oggetti è:

1. Identificare i componenti.
2. Definire l'interfaccia dei componenti.
3. Definire le modalità con cui le interfacce consentono l'interazione fra oggetti.
4. Minimizzare le relazioni fra i componenti.

Generalmente, è buona pratica documentare il programma ad ogni sviluppo, ma con grandi quantità di dati, la cosa può risultare tediosa. Per questa ragione si usa uno strumento utile per la documentazione delle classi: l'**Unified Modeling Language**, il quale consente di definire graficamente e quindi documentare un sistema orientato agli oggetti. Verrà approfondito più avanti nel corso.

1.2 Il linguaggio 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 essere compilato, 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** controlla eventuali leaks di memoria per tapparle.
- 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, tipico se vuoi giocare a Minecraft, 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.

Si suppone che tu abbia frequentato il precedente corso di programmazione, tenuto con C. Conoscerlo faciliterà enormemente le cose, poiché la sintassi per le funzionalità base

del linguaggio ne è in gran parte simile. Infatti si mantengono i tipi primitivi: **int**, **double**, **char**, aggiungendo **boolean** e **String**, quest'ultimo non primitivo, ma successivamente approfondito. Si mantengono inoltre altre funzionalità come il type casting. Riassumendo, diciamo che la sintassi base di Java è 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.

La vera novità di nostro interesse sono invece le classi e gli oggetti; lo scopo di questi concetti è l'astrazione, un modo per rendere più vicino al nostro pensiero la programmazione.

Supponiamo che una classe rappresenti gli animali, con delle caratteristiche generali come nome e verso, detti **campi**; a questo punto è possibile istanziare un oggetto con nome "cane" che esegue il verso "woof". La programmazione orientata agli oggetti si basa su questo workflow, espandibile con ulteriori concetti che andremo a vedere più avanti nella dispensa.

Chapter 2

Funzioni OOP di Java

2.1 Struttura ed esecuzione dei programmi

Java è usato tendenzialmente per la gestione di programmi di medie dimensioni, quindi necessariamente suddivisi in più parti. Il compilatore vede ogni file con estensione ".java" come una singola classe, ed infatti è richiesto che il nome associato alla classe nel file debba essere uguale a quello del file effettivo. Al sui interno bisognerà poi dichiarare un metodo di nome "main", che fungerà da entry point per quel singolo file. Compilato il programma, sarà possibile eseguirlo grazie al launcher.

```
1  // Per compilare: javac HelloWorld.java
2  // Per eseguire: java HelloWorld
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 }
```

Il compiler restituisce in output un file di estensione ".class". Infatti, ogni file è visto come classe con i relativi oggetti. Il caricamento nella compilazione di queste classi è basato sul **classpath**, la lista di locazioni dove queste possono essere prese. Se il compiler non trova la classe, lancerà un'eccezione e non verrà creato l'eseguibile.

Ci sono più modi per indicare un classpath al compilatore; indicarla singolarmente oppure contenere tutte le classi in un barattolo, un file con estensione ".jar". Questi sono fondamentalmente degli archivi compressi in cui è salvato il bytecode e altre meta-informazioni. Quindi:

- Per eseguire da cartelle diverse: $ java -cp ./nomeCartella nomeEseguibile
- Per creare un file .jar: $ jar cvf nomeJar.jar classOne.class classTwo.class ...

- Per eseguire un programma con file .jar: $ java -cp nomeJar.jar nomeEseguibile

Nella creazione di un file .jar è possibile aggiungere anche una direttiva su quali classi eseguire di preciso e quali ignorare. Ciò si fa con un file nominato **manifest.txt**.

```
// manifest.txt
Main-Class: nomeClasse

// Compilazione
jar cvfm nomeJar.jar manifest.txt classOne.class
```

Notare i termini **cvf** e **cvfm**, significanti rispettivamente create verbose file e create verbose file manifest.

2.2 Dichiarazione di classi e oggetti

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

```
// Dichiarazione della classe Date
public class Date {

  // Scope dei campi della classe, questi sono attributi...
  int day, month, year;

  // ...e questo un metodo.
  String printDate() {
    return day + "/" + month + "/" + year;
  }
}
```

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:

```
// Funzione costruttore. Qui popola le variabili dell'oggetto.
 Modificatore public.
public Date(int day, int month, int year) {
  // "this." indica la locazione dell'oggetto corrente.
  this.day = day;
  this.month = month;
  this.year = year;
}

// 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:

```
1  public class MainDate {
2    public static void main(String[] args) {
3
4      // Utilizzo di isFirstDay fuori oggetto. Possibile dato static.
5      System.out.println(Date.isFirstDay(1));
6
7      // Creazione dell'oggetto today e stampa con il metodo printDate.
8      Date today = new Date(14, 10, 2024);
9      System.out.println(today.printDate());
10   }
11 }
```

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.

2.3 Librerie utili

Le librerie digitali di Java sono un insieme di funzioni contenute in files appositi e sono fornite da terze parti. Come le hanno fatte gli altri utenti, pure te potresti scriverne. Alla fine, tutti i file da eseguire sono visti come classi, ed infatti dovranno essere comprese nel classpath, se non sono standard. Un pacchetto di librerie base che è comodo imparare è dato da:

- **java.lang**: La libreria nativa del linguaggio. Comprende tutte le funzioni di base come oggetti e costrutti.
- **java.io**: Fornisce funzioni relative alla gestione di datastream in input e output.
- **java.math**: Usata per il calcolo matematico.
- **java.awt**: Usata per la gestione di interfacce utente e grafiche.

- **java.util**: Contiene varie funzioni di utility il cui vero valore sarà compreso andando avanti nel corso.

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. In codice:

```
// Aggiunge la classe Scanner da util. Segnatura generale: [java.nomeLibreria.nomeClasse]
import java.util.Scanner;

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

    // Dichiarazione dell'oggetto keyScan di classe Scanner
    Scanner keyScan = new Scanner(System.in);
    int n1, n2, res;

    // print al posto di println stampa senza andare a capo alla fine.
    System.out.print("Inserisci il primo fattore: ");
    // Assegna a n1 l'intero letto da tastiera, idem n2.
    n1 = keyScan.nextInt();
    System.out.print("Inserisci il secondo fattore: ");
    n2 = keyScan.nextInt();

    // Chiudi lo scanner con il metodo close().
    keyScan.close();

    // Stampa della variabile concatenata alla stringa "Risultato: "
    System.out.println("Risultato: " + n1*n2);
  }
}
```

Continuiamo ragionando sull'ultima istruzione dello snippet appena visto. Le stringhe, come già detto, sono viste come oggetti, e per usarle bisogna istanziare un oggetto. Dopodiché, non saranno più mutabili in quanto non sono viste come sequenze di caratteri. Tuttavia, abbiamo a disposizione alcuni metodi con cui lavorarci:

```
String str = new String("SHAW!");
String same = new String("SHAW!");

System.out.println(str.length());   // Stampa la lunghezza di una stringa.
System.out.println(str.charAt(4));    // Stampa il singolo carattere alla posizione 4.
System.out.println(str.equals(same)); // Confronta str e same, torna vero se sono uguali.
str.indexOf(a)              // Ricerca del carattere 'a' all'interno della stringa. Se assente torna -1, altrimenti torna l'indice.
```

```
str.substring(start, end)       // Ottiene una sottostringa con gli
 indici dati in input. Crea una nuova stringa.
str.replace(target, replacement)    // Rimpiazza i caratteri di
 target con quelli di replacement.
```

La classe String possiede anche il metodo **format**. Permette di formattare la stringa con una sintassi simil-C, ed è utile per mantenere un template.

```
String str = new String("Egale");
int num = 2;

// Dona un formato alla stringa
String formatStr = String.format("%s = %d", str, num);
System.out.println(formatStr);
```

Sebbene gli oggetti da String diventino costanti post-inizializzazione, java.lang fornisce la classe **StringBuilder**, che consente di gestire dinamicamente una stringa.

```
StringBuilder sb = new StringBuilder("Droxie");       // Crea una
 stringa mutevole
sb.append("kaliemu");                   // Concatena in coda i
 caratteri fra le virgolette.
```

È possibile anche cancellare ed inserire sottostringhe con i metodi **delete** ed **insert**. Insomma, le stringhe base sono costanti, mentre le stringhe mutevoli di StringBuilder sono sicuramente più flessibili.

2.4 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.

2.5 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

Pratiche corrette e gestione errori

Chapter 4

Documentazione del codice