

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

1 Introduzione	4
1.1 Programmazione Orientata agli Oggetti	4
1.2 Funzionamento di Java	5
1.3 Struttura, compilazione ed esecuzione dei programmi	6
1.4 Gestione di un progetto su più file	6
1.5 SPOSTARE: Il linguaggio Java	7
2 Il linguaggio Java	9
2.1 Dichiarazione di classi e oggetti	9
2.2 Librerie utili	10
2.3 Incapsulamento e information hiding	12
2.4 Array, matrici e tipo enum	13
3 Librerie utili	15
3.1 java.lang.*	15
3.1.1 java.lang.System	15
3.1.2 java.lang.String	15
3.1.3 java.lang.Object	15
3.1.4 java.lang.Integer, java.lang.Character	15
3.1.5 java.lang.AutoCloseable	15
3.1.6 java.lang.Iterable<T>	15
3.1.7 java.lang.Thread	15
3.2 java.util.*	15
3.2.1 java.util.Scanner	15
3.2.2 java.util.Arrays	15
3.2.3 java.util.Iterator<T>	15
3.2.4 Strutture dati	15
3.3 java.io.*	16
3.3.1 java.io.Closeable	16
3.3.2 java.io.Reader, java.io.FileReader, java.io.BufferedReader	16
3.3.3 java.io.Writer, java.io.FileWriter, java.io.BufferedWriter, java.io.PrintWriter	16

<i>CONTENTS</i>	3
4 Boh, da vedersi	17

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.

```

1 // Compila: javac -d bin/ src/pkg/MyClass.java
2 // Impacchetta: jar cvf MyJar.jar -C bin/ pkg/
3 // Esegui: java -cp MyJar.jar pkg.MyClass
4
5 // Segnala che la classe MyClass sta all'interno del pacchetto pkg.
6 package pkg;
7
8 public class MyClass {
9     public static void main(String[] args) {
10         // Blocco di codice
11     }
12 }
```

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.

```

1 // Impacchetta: jar cvfm MyJar.jar manifest.txt -C bin/ pkg/
2 // Esegui: java -jar MyJar.jar
3
4 // manifest.txt
5 Main-Class: pkg.MyClass
```

1.5 SPOSTARE: Il linguaggio Java

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.
- **Identifieri:** 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

Il linguaggio Java

2.1 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 costruttrice**. 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 costruttrice. Qui popola le variabili dell'oggetto.
2     Modificatore public.
3     public Date(int day, int month, int year) {
4         // "this." indica la locazione dell'oggetto corrente.
5         this.day = day;
6         this.month = month;
7         this.year = year;
8     }
```

```

9 // Metodo isFirstDay con modificatore static. Non ha accesso a campi
10 non statici.
11 static boolean isFirstDay(int day) {
12     if (day == 1) return true;
13     else return false;
14 }
```

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.

2.2 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:

```

1 // Aggiunge la classe Scanner da util. Segnatura generale: [java.
2   nomeLibreria.nomeClasse]
3 import java.util.Scanner;
4
5 public class Mult {
6   public static void main(String args[]) {
7
8     // Dichiarazione dell'oggetto keyScan di classe Scanner
9     Scanner keyScan = new Scanner(System.in);
10    int n1, n2, res;
11
12    // print al posto di println stampa senza andare a capo alla fine
13    System.out.print("Inserisci il primo fattore: ");
14    // Assegna a n1 l'intero letto da tastiera, idem n2.
15    n1 = keyScan.nextInt();
16    System.out.print("Inserisci il secondo fattore: ");
17    n2 = keyScan.nextInt();
18
19    // Chiudi lo scanner con il metodo close().
20    keyScan.close();
21
22    // Stampa della variabile concatenata alla stringa "Risultato: "
23    System.out.println("Risultato: " + n1*n2);
24  }
}
```

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:

```

1 String str = new String("SHAW!");
2 String same = new String("SHAW!");
3
4 System.out.println(str.length());    // Stampa la lunghezza di una
5   stringa.
6 System.out.println(str.charAt(4));    // Stampa il singolo carattere
7   alla posizione 4.
8 System.out.println(str.equals(same)); // Confronta str e same, torna
9   vero se sono uguali.
```

```

7 str.indexOf(a)           // Ricerca del carattere 'a' all'interno
8   della stringa. Se assente torna -1, altrimenti torna l'indice.
9 str.substring(start, end) // Ottiene una sottostringa con gli
  indici dati in input. Crea una nuova stringa.
10 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.

```

1 String str = new String("Egale");
2 int num = 2;
3
4 // Dona un formato alla stringa
5 String formatStr = String.format("%s = %d", str, num);
6 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.

```

1 StringBuilder sb = new StringBuilder("Droxie");           // Crea una
  stringa mutevole
2 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.

2.3 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.

```

1 public class Person {
2     // Dichiarare senza modificatori setta public di base
3     String name;
4     private int age = -1;
5
6     Person(String name) { this.name = name; }
7
8     // Metodo per segnalare se age risulta illegale
9     void changeAge(int age) {
10         if (age > 0 && age <= 122) this.age = age;
11         else System.out.println("Invalid age.");

```

```
12     }
13 }
```

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.

2.4 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.

```
1 // Dichiarazione di array
2 int[] arr = new int[dimensione];
3 // Scrittura in un indice specifico
4 arr[numeroIndice] = NomeClasse(eventuali_parametri);
5 // Accesso ad un elemento in un indice specifico
6 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 sintassi più compatta.

```
1 // Enhanced for loop, equivalente alla scrittura classica
2 for (tipoVariabile : espressioneIterabile);
3 // Confronta se due array hanno gli stessi elementi
4 Arrays.equals(arr1, arr2);
5 // Stampa ogni elemento dell'array in formato stringa
6 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.

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

```
4 int[] tmp = matrix[0];
5 matrix[0] = matrix[matrix.length-1];
6 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 `java.lang.*`

- 3.1.1 `java.lang.System`
- 3.1.2 `java.lang.String`
- 3.1.3 `java.lang.Object`
- 3.1.4 `java.lang.Integer, java.lang.Character`
- 3.1.5 `java.lang.AutoCloseable`
- 3.1.6 `java.lang.Iterable<T>`
- 3.1.7 `java.lang.Thread`

3.2 `java.util.*`

- 3.2.1 `java.util.Scanner`
- 3.2.2 `java.util.Arrays`
- 3.2.3 `java.util.Iterator<T>`

3.2.4 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>`

3.3 java.io.*

3.3.1 java.io.Closeable

3.3.2 java.io.Reader, java.io.FileReader, java.io.BufferedReader

**3.3.3 java.io.Writer, java.io.FileWriter, java.io.BufferedWriter,
java.io.PrintWriter**

Chapter 4

Boh, da vedersi