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, 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**. Ciò vale sia per variabili che per oggetti.

2.2 Dichiarazione di classi e oggetti, this, parametri formali e attuali

Le classi e gli oggetti sono il fulcro sul quale si basa la programmazione in Java ed il paradigma a oggetti in generale. Come già menzionato, le prime si possono vedere come

un insieme, il quale detiene attributi e metodi, ed i secondi sono le singole istanze delle classi con le suddette specifiche. Data questa struttura, ne consegue che per lavorare dovremo prima definire una classe, per poi istanziare gli oggetti.

```
// Dichiarazione della classe Date nel file Date.java
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;
  }
}
```

```
// Dichiarazione della classe MainDate nel file MainDate.java
public class MainDate {
  public static void main(String[] args) {
    // Istanziazione dell'oggetto Today
    Date Today = new Date();
  }
}
```

Come si può vedere, la classe rappresenta il corpo contenente tutto il codice, mentre gli oggetti vengono istanziati in un altro file, ove richiesto, con la keyword **new**. Questa istruzione fa sì che venga allocato spazio sufficiente per contenere i dati dell'oggetto.

I metodi dichiarati possono, ovviamente, anche ricevere parametri, per la loro eventuale gestione ed elaborazione. Possiamo quindi introdurre il concetto di **parametri formali** e **parametri attuali**. I primi sono quelli dati all'inizio del metodo, mentre i secondi sono il risultato dell'elaborazione della funzione e sarà ciò che viene ritornato.

```
// Il metodo prende il parametro formale n
public int incr (int n) {
  // res e' il parametro attuale elaborato
  int res = n+1;
  return res;
}

public void printIncr (int x) {
  int y = incr(x);
  System.out.println(y);
}
```

Bisogna quindi prestare attenzione a quale dato viene ritornato dal metodo, perché se nella funzione incr ritornassimo n, tutta l'elaborazione sarebbe stata inutile.

Un'altro concetto molto importante è la parola chiave per far riferimento all'oggetto corrente: la keyword **this**, la quale può essere usata per attributi e variabili ed è particolarmente utile per evitare ambiguità fra gli oggetti.

```java
public class Cliente {
  private String nome;

  // Il metodo prende il nome dato come parametro
  public void getName (String nome) {
    // Associa il parametro al preciso oggetto corrente
    this.nome = nome;
  }
}
```

L'utilizzo di this è molto importante per rendere il codice chiaro ed è uno standard da rispettare. In assenza di this, il programma non avrebbe capito quale stringa nome prendere, causando errori di compilazione.

2.3 Metodo costruttore e overloading

I **metodi costruttore** sono funzioni di una classe particolari, il cui blocco di codice determina ciò che viene eseguito all'istanziazione di un oggetto. In assenza di dichiarazione di tale metodo, il compilatore aggiungerà quello di default, che non esegue niente.

Ogni classe ha quindi un costruttore, il quale avrà il suo stesso nome e potrà prendere parametri, ma sarà sprovvisto della keyword return.

```java
public class Punto {
  int x, y;

  // Metodo costruttore. Stampa la stringa ad istanziazione.
  public Punto () {
    System.out.println("Punto istanziato");
  }
}
```

È possibile inoltre definire più costruttori in una classe, ma dovranno essere differenziati in base alla segnatura, quindi ai parametri. Per capire quale prendere, introduciamo il concetto di **constructor overloading**.

Supponiamo di definire un costruttore "Punto(int x)"; nell'istanziazione dell'oggetto, se sarà dato un intero come parametro, il compilatore sceglierà quest'ultimo, in alternativa, se non è dato niente, sceglierà il primo. Qualunque altro scenario comporta errori di compilazione.

2.4 Modificatori, visibilità e memoria di un programma

Avrai sicuramente notato le keyword usate nella dichiarazione di classi o utilizzate per le variabili; queste si chiamano **modificatori** e sono capaci di cambiare il significato delle

componenti del programma. Approfondiamone il concetto ed il significato.

Modificatore public
Applicabile a classi, oggetti e variabili. Rende la componente accessibile in qualunque altro file. Tendenzialmente, volendo rendere il codice sicuro, non viene utilizzato per variabili.

```
public class PublicClass {
  public PublicClass () { /* Blocco di codice */ }
}
```

Modificatore protected
Applicabile a oggetti, metodi e attributi. Restringe l'accesso alla componente, consentendolo solo alla classe in cui è dichiarata e le sue eventuali sottoclassi, a prescindere dal pacchetto. Attua il principio di ereditarietà, il quale sarà spiegato in dettaglio più avanti.

```
public class PublicClass {
  protected int value = 1;
}

public class TrustedPerson extends PublicClass {
  // La subclass accede a value senza problemi
  int safeVar = value;
}
```

Modificatore private
Applicabile a oggetti e attributi. Rende la componente visibile esclusivamente dalla classe in cui è dichiarata. Usato molto spesso per sfruttare l'incapsulamento, se vi si prova ad accedere da classi differenti, il compiler darà errore.

```
public class PublicClass {
  private int invisible = 1;
}

public class NoobHacker {
  // Azione illegale. Il compiler si lagna.
  int stealVar = invisible;
}
```

Modificatore package private
Assegnato in automatico dal compilatore in assenza di modificatori, restringe l'accesso, consentendolo esclusivamente a classi di uno stesso pacchetto.

Prima di introdurre l'ultimo modificatore, è necessario parlare della memoria utilizzata dai programmi Java. Abbiamo una parte di memoria **static**, usata per elementi condivisi da tutte le istanze di classe, come definizioni di classe e campi statici; un'altra di

heap, usata per l'allocazione di memoria dinamica e quindi istanziazione di oggetti, ed un'ultima di **stack**, dedita agli elementi creati in runtime. Tradotto in codice, abbiamo:

```
public class memoryTest {
  // Variabile nuova, spazio allocato in heap
  int i;
  // Oggetto String statico, condiviso da ogni classe
  static String s1;
  void aMethod () {
    // s2 non static, elemento creato in runtime, va in stack memory.
    String s2 = new String("abc");
    s1 = s2;
  }
}
```

Nota importante da tenere a mente: i metodi statici possono interagire esclusivamente con altri metodi static; altrimenti si avrà un errore di compilazione.

2.5 Costrutti condizionali e ciclici

Ogni linguaggio di programmazione contiene i costrutti per la manipolazione del flow, siano essi condizioni o cicli non importa. Saranno sempre presenti, seppur con forma differente. Supponendo che tu abbia già frequentato programmazione 1, salterò direttamente alle sintassi diverse da quelle del C, perché **if-else**, **while**, **do-while**, **switch** e **for** sono definiti esattamente come nel suddetto linguaggio.

Tuttavia, dalla versione Java 5, è stato introdotto l'**enhanced for loop**, che svolge esattamente allo stesso modo le funzioni del suo gemello, ma ha una sintassi semplificata.

```
for (tmpVar : iterObj) {
  // Blocco di istruzioni
}
```

In parole povere, dice "scorri tutto iterObj con indice tmpVar", quindi è consigliabile usarlo quando si ha la certezza di voler scorrere un oggetto iterabile per intero.

Anche il costrutto switch ha ottenuto migliorie. Mettiamo caso di avere più casi che riportano uno stesso numero; in una scrittura tradizionale sarebbe necessario definire casi separati, ma in Java è possibile raggrupparli ed usare un operatore **arrow** per indicare ciò che devono eseguire.

```
switch (colore) {
  case VERDE -> System.out.println("Luce verde");
  case GIALLO -> System.out.println("Luce gialla");
  case ROSSO -> System.out.println("Luce rossa");
}
```

Non solo, ma possiamo anche ritornare valori dal blocco con la keyword **yield**, per il quale abbiamo due scritture diverse. Una usa arrow, l'altra i due punti e dove entrambe svolgono una stessa funzione, non è possibile mischiarle. Bisogna necessariamente scegliere

una delle due notazioni. Questo perché usando la prima, è come se ci fosse un break incorporato nel caso, mentre nella seconda è ancora presente il fall-through.

```
// Notazione con operatore arrow '->'
String season = switch (month) {
  case DECEMBER, JANUARY, FEBRUARY -> "Winter";
  case MARCH, APRIL, MAY -> "Spring";
  case JUNE, JULY, AUGUST -> "Summer";
  case SEPTEMBER, OCTOBER, NOVEMBER -> "Autumn";
}

// Notazione con operatore colon ':'
String season = switch (month) {
  case DECEMBER, JANUARY, FEBRUARY: "Winter";
  case MARCH, APRIL, MAY: "Spring";
  case JUNE, JULY, AUGUST: "Summer";
  case SEPTEMBER, OCTOBER, NOVEMBER: "Autumn";
}
```

2.6 Array, enumerazioni e record

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 e agevolata di array è utile la libreria **java.util.Arrays**, contenente metodi per effettuare confronti e stampare il contenuto. Verrà approfondita più avanti. Naturalmente, per scorrerli e lavorarci sarà necessario utilizzare costrutti ciclici.

Le matrici hanno ricevuto delle migliorie; la loro dichiarazione è simile a quella degli array ed è possibile usare su di esse tutti i metodi utili per questi ultimi. Inoltre, in quanto sequenze di puntatori, è possibile scambiare facilmente righe e colonne.

```java
// Dichiarazione ed inizializzazione di una matrice di interi
int[][] matrix = {{1,2,3}, {4,5,6}, {7,8,9}};
// Per scambiare righe o colonne puoi prendere il puntatore.
int[] tmp = matrix[0];
matrix[0] = matrix[matrix.length-1];
matrix[matrix.length-1] = tmp;
// Stampa gli elementi di una matrice
for (int[] row : matrix) System.out.println(Arrays.toString(row));
```

Un altro elemento importante per la programmazione in Java sono i **tipi enumerazioni**. Sono classi che rappresentano un insieme finito di costanti, le quali sono tutte le istanze che può assumere l'eventuale oggetto.

Le enumerazioni si definiscono con la keyword **enum**, ed il loro vantaggio sta nell'efficienza del processo di recupero valori; è infatti consigliato utilizzarle in necessità di richiamare più volte determinati dati.

```java
// Dichiarazione dell'enumerazione
public enum Numbers { ONE, TWO, THREE };
// Istanziazione dell'oggetto
Numbers numOne = Numbers.ONE;
```

Gli elementi delle enumerazioni, in quanto già rappresentanti un'istanza della classe, sono implicitamente del tipo Months, per questo non è necessario utilizzare new

Chapter 3

Paradigmi OOP

3.1 Incapsulamento

L'incapsulamento è un concetto che ha origine nelle modalità di interazione nel mondo reale. Così come per interagire con una persona si usa la voce per ottenere informazioni specifiche, allo stesso modo gli oggetti comunicano con interfacce apposite al fine di ritornare dati. Questa filosofia si ottiene controllando l'accesso ai dati tramite le keyword dei modificatori di accesso. Più precisamente, bisogna:

1. Dichiarare gli attributi di una classe come **privati**, per renderli inaccessibili ad altre classi al di fuori di questa.

2. Creare dei metodi **pubblici** per accedere, modificare e riportare gli attributi privati in maniera controllata, talvolta con dei controlli per assicurarsi che i dati siano presentati con certe caratteristiche. Questi metodi si chiamano **setter**, per assegnare il dato all'oggetto, e **getter**, per accedere al dato dell'oggetto.

```
1  public class Date {
2    private int day, month, year;
3
4    // Metodo setter per garantire che giorno sia un valore corretto
5    public void setDay (int g) {
6      if (j >= 1 && j<= 31) giorno = g;
7      else System.out.println("Invalid day");
8    }
9
10   // Metodo getter per ritornare il valore in sicurezza
11   public int getDay () { return day; }
12
13   public void setMonth (int m) {
14     if (m <= 1 && m <= 12) month = m;
15     else System.out.println("Invalid month");
```

```
16      }
17
18      public int getMonth () { return month; }
19
20      public void setYear (int y) {
21        if (y <= 0) year = y;
22        else System.out.println("Invalid year");
23      }
24
25      public int getYear () { return year; }
26    }
```

3.2 Ereditarietà

- Definizione di classi per estensione, ereditando campi e metodi - Relazione superclasse/-sottoclasse e concatenazione dei costruttori da sottoclasse a superclasse con super() - Ereditarietà singola e multipla

3.3 Polimorfismo

- Ridefinizione o aggiunta di metodi

3.4 Astrazione

- Interfaccia pubblica di una classe - Interfacce e relativo utilizzo - Metodi e classi abstract - Aliasing fra variabili e rischi

Chapter 4

Librerie utili

4.1 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 }
```

4.2 java.lang.*

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

4.2.1 Classi wrapped

4.2.2 .System

Costanti err, in, out, metodo currentTimeMillis().

4.2.3 .String, StringBuilder

.String: Le stringhe son costanti e senza terminatore, str.length(), str.charAt(i), s.equals(t), str.indexOf(c), str.substring(start, end), str.replace(target, replacement), format()

.StringBuilder StringBuilder delete(int start, int end), StringBuilder insert(int index, String str), StringBuilder reverse()

4.2.4 .Math

4.3 java.util.*

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

4.3.1 .Scanner

netxLine, nextInt, nextFloat

4.3.2 .Random

4.3.3 .Arrays

4.4 java.io.*

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