1 Lezione del 23-09-25

1.1 Introduzione

Il corso di programmazione avanzata si pone di approfondire gli aspetti di programmazione *orientata agli oggetti* (**OOP**) e *concorrente*, sfruttando sia come strumento che come fine il linguaggio di programmazione **Java**. Parleremo sia di programmazione desktop che applicazioni in rete (cioè servizi per pagine Web).

1.1.1 Cenni storici

Java nasce nei primi anni '90 come *Oak* all'interno della *Sun Microsystems* (poi acquisita da Oracle).

L'idea originale del progetto era quella di realizzare un linguaggio per la programmazione di dispositivi di elettronica di consumo. Per questo motivo una delle prime prerogative del progetto era che il linguaggio potesse creare programmi che giravano su una variegata gamma di piattaforme hardware (il cosiddetto paradigma **WORA**, *Write Once, Run Anywhere*).

Con l'avvento del Web e delle applicazioni distribuite in modello client-server, l'obiettivo del progetto virò verso la creazione di un linguaggio che potesse essere eseguito sui client, all'interno dei browser (nei cosiddetti *applet*), a scapito dell'architettura o il S/O locale (browser Netscape su architetture non-Intel e S/O Unix come browser Explorer su architetture Intel e S/O Microsoft).

Oggi questo tipo di uso è in declino, ma il linguaggio Java rimane estremamente popolare per lo sviluppo di applicazioni desktop/mobile, e in contesto sopratutto aziendale per lo sviluppo di applicazioni lato server.

1.1.2 Versioni

Possiamo individuare alcune versioni di rilevo dello standard Java.

- Nel 1996 viene lanciata la JDK 1.0 (*Java Development Kit*), prima versione del linguaggio;
- Nel 1998 viene lanciata la J2SE 1.2, da cui la denominazione *Java 2*, rimasta fino alla 5.0 del 2004 (fra l'altro slegata in nome dalla versione precedente, la 1.4);
- Dal 2006 vengono usate le denominazioni Java SE 6 e successive, ed è stato adottato un modello di rilascio periodico (da Java SE 9 annuale). Questo è più o meno in corrispondenza dell'acquisto da parte di Oracle di Sun Microsystems, avvenuto nel 2010 (la prima versione lanciata sotto la Oracle fu Java SE 7).

Nel progetto originale della Sun Microsystems erano previste più *edizioni* di Java, fra cui:

- Java **ME**, cioè *Micro Edition*, che mirava a piattaforme con risorse limitate;
- Java **SE**, cioè *Standard Edition*, che mirava a piattaforme desktop;
- Java **EE**, cioè *Enterprise Edition*, che mirava a piattaforme distribuite e in rete.

Java è fornito sia come **JDK** (come già detto *Java Development Kit*), comprensivo di compilatore e strumenti di sviluppo, che come **JRE** (*Java Runtime Environment*), pensato solo per l'esecuzione di applicazioni già sviluppate.

Notiamo poi che noi useremo la piattaforma **OpenJDK**, implementazione open source del JDK lanciata da Sun Microsystems nel 2006, prima dell'acquisizione da parte di Oracle, e ancora oggi mantenuta da Oracle ma comunque mantenuta open source.

1.1.3 Natura del Java

I design goal di Java sono i seguenti:

- Semplice, paradigma orientato agli oggetti e familiare per gli sviluppatori di linguaggi precedenti (si pensi alla famiglia C/C++, con cui condivide sintassi e diversi costrutti);
- Robusto e sicuro, in particolare riguardo alla memoria (non sono presenti meccanismi come i puntatori, e la gestione della memoria è quindi *sicura* e largamente fuori dalle mani del programmatore);
- *Architecture-neutral*, cioè neutrale all'architettura e portatile su una vasta gamma di piattaforme (come già detto, paradigma *WORA*). In questo, la specifica non contiene ambiguità (i tipi, ad esempio, sono standardizzati in dimensioni, a differenza del C++ che è *platform-dependent*).

  Il codice sorgente Java viene compilato nel cosiddetto **bytecode**, una sorta di codice macchina che viene eseguito sulla **JVM**, una macchina virtuale basata sullo *stack* piuttosto che su *registri*, e agnostica al livello fisico sottostante;
- Alte prestazioni, difficili da ottenere a causa della sua natura sostanzialmente interpretata, ma comunque quasi paragonabili nelle ultime versioni a codice scritto con linguaggi come C/C++, sopratutto nel caso di codice che viene ripetuto molte volte. Questo è reso possibile anche dall'uso di tecnologie di compilazione **JIT** (compilazione *Just In Time*).

  Come nota storica, vediamo che sono state sviluppate implementazioni hardware della JVM, oggi non più particolarmente in voga;
- Interpretato, *threaded*, cioè ottimizzato per l'esecuzione su sistemi *multithreaded*, e dinamico per quanto riguarda il collegamento delle librerie, che viene effettuato a tempo di esecuzione.

Abbiamo quindi che le differenza principali fra i linguaggi C/C++ a cui siamo abituati e il Java sono:

- La natura dinamica del Java (anche le classi possono essere caricate a tempo di esecuzione);
- Il supporto nativo per il multithreading, che in Java è praticamente immediato, mentre in C++ richiede API e tecniche di programmazione considerevolmente più complesse.

1.1.4 Java e Android

Un caso di applicazione di Java degno di nota è quello dello sviluppo di applicazioni per il sistema operativo Android.

Android era infatti fornito nelle prime versioni della macchina virtuale **Dalvik**, basata sui registri, che esegue codice Java compilato in un bytecode apposito, appunto il codice *Dalvik* (da cui `.dex`, *Dalvin EXecutable*).

Il supporto per la macchina Dalvik è rimasto in Android fino alla versione 4.4 *KitKat*, ed è stato seguito nella 5.0 *Lollipop* da **ART** (*Android RunTime*), che usa lo stesso bytecode e gli stessi eseguibili, ma con diverse ottimizzazioni.

Notiamo poi che oggi (dal 2019) Android è una piattaforma *Kotlin-first*, cioè orientata al linguaggio "successore" del Java, il **Kotlin**.

1.2 Ciao mondo in Java

Possiamo quindi vedere il nostro primo programma di esempio in Java.

```
class Main {
  public static void main(String[] args) {
    System.out.println("C# is better");
  }
}
```

Vediamo come tutto sta necessariamente dentro una classe, qui la classe `Main`, che definisce un metodo, qui il metodo `main()`.

Questo metodo è specifico (come il `main()` del C/C++), viene eseguito quando la classe che lo possiede è invocata come programma, e ha come argomento la lista degli argomenti programma `String[] args`, dove le quadre definiscono un'array come per la normale sintassi c-like. Inoltre, il metodo `main()` è dichiarato come:

- `public`, cioè chiunque può invocarlo;
- `static`, cioè appartiene alla definizione di classe stessa, e non ad una particolare istanza di classe.

Da qui usiamo la funzione di libreria `println` per stampare un messaggio a video.

Possiamo compilare questo codice inserendolo in un file col nome della classe definita, `Main.java`, e compilare come:

```
javac Main.json
```

Questo creerà un nuovo file, `Main.class`, contenente appunto la classe `Main`. A questo punto si esegue come:

```
java Main
```

notando che si riporta solo il nome della classe, senza l'estensione `.java`.

1.3 Tipi del Java

Il Java è un linguaggio fortemente tipizzato dove ogni oggetto ha un tipo. In particolare notiamo fra tipi **primitivi** e di **riferimento**.

- I tipi **primitivi** sono i classici tipi di dato disponibili negli altri linguaggi:

```
int x;
float y = 5.6;
double pippo = 3.2 + Math.sqrt(7.4);
int i1, i2, i3;
char a = 'a';
// String s = "Also try Kotlin";
/* const int VAL; no ! */
static final int VAL; // e' effettivamente costante
```

La lista completa dei tipi è la seguente:

Tipo	Descrizione
`boolean`	Un booleano `true` o `false`
`char`	Un carattere, cioè un *codepoint* UTF-16 su 16 bit
`byte`	Un intero su 8 bit
`short`	Un intero su 16 bit
`int`	Un intero su 32 bit
`long`	Un intero su 64 bit
`float`	Un numero in virgola mobile su 32 bit come definito da IEEE 754
`double`	Un numero in virgola mobile su 64 bit come definito da IEEE 754

Notiamo innanzitutto che il Java adotta la codifica UTF-8 per le stringhe e UTF-16 per i caratteri. Ricordiamo poi che, come già detto, i tipi del Java (ad esempio i tipi numerici) hanno dimensione fissata dalla specifica e non *platform-specific*.

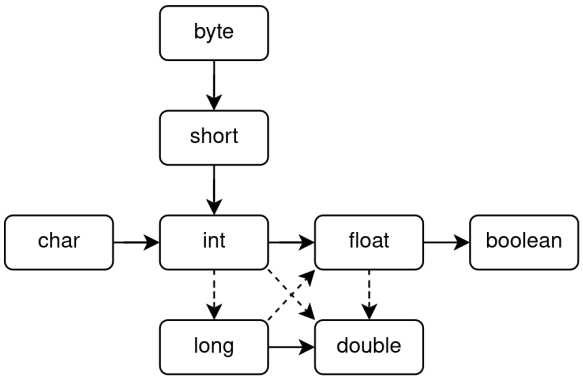
Vediamo poi che la parola chiave `const` non è implementata in Java (anche se è comunque riservata), mentre sono disponibili le parole chiave `static` e `final`. In particolare questa assegna a variabili valori finali, cioè che non possono più essere alterati.

I valori non inizializzati in Java vengono lasciati in uno stato `undefined`, possono restare tali fino alla definizione, ma se si prova ad usarli prima il compilatore lancia un errore.

Per ogni tipo primitivo esiste poi un corrispondente *tipo oggetto*, cioè sostanzialmente una classe *wrapper* per quel tipo di primitivo.

Conversioni fra tipi

La conversione fra tipi primitivi e oggetto corrispondenti è automatica. Sono poi previste le conversioni implicite fra tipi primitivi secondo i meccanismi che ci aspettiamo:



Per effettuare cast nelle direzioni opposte a quelle previste dalle conversioni implicite si usano le conversioni esplicite secondo la solita sintassi:

```
float pi = 3.14;
int engineer_pi = (int) pi; // engineer_pi = 3
```

- I tipi di **riferimento** comprendono:
  - Array;
  - Classi;
  - Interfacce.

1.3.1 Costrutti del Java

Il Java comprende tutti i costrutti condizionali e di ripetizione a cui siamo abituati. Notiamo in particolare la presenza del costrutto *for-each* disponibile come:

```
for(tipo variabile : espressione_insieme) {
  // corpo
}
```

che itera sugli elementi dell'oggetto `set_expression` instanziandoli ogni volta nell'oggetto `variabile`.

Vediamo quindi tutti i costrutti disponibili:

Costrutto	Sintassi
if-else	`if(espressione) { // vera } else { // falsa }`
switch	`switch(espressione) { case a: // caso a case b: // caso b // ... default: // caso default }`
while	`while(espressione) { // corpo }`
do-while	`do { // corpo } while(espressione);`
for	`for(espressione; espressione; espressione) { // corpo }`
for-each	`for(tipo variabile : espressione_insieme) { // corpo }`
break	`break;`
continue	`continue;`

1.3.2 Classe `System`

`System` è una classe delle librerie Java dotata di 3 campi per la gestione degli stream di I/O:

- `System.in`, che gestisce il flusso di ingresso (tastiera). Per l'ingresso inglobiamo lo stream `System.in` all'interno della classe `Scanner` come segue:

```
Scanner s = new Scanner(System.in);
```

  A questo punto possiamo usare i metodi della classe scanner come `next()` (che si ferma alla parola) e `nextLine()` (che si ferma alla riga) per leggere dallo stream. Vediamo che la classe `Scanner` è la prima che incontriamo a non essere inclusa di default nel sorgente, ma che va inclusa con la direttiva `import`:

```
import java.util.Scanner;

// classe main, ecc...
```

- `System.out`, che gestisce il flusso di uscita (video). Sono disponibili i metodi `println()`, e `print()` (che non stampa i caratteli di ritorno carrello e nuova linea);
- `System.err`, che gestisce il flusso di errore (sempre video);

1.4 Classi del Java

In Java *tutto* il codice è dentro classi. Una classe contiene:

- La definizione della struttura degli oggetti (istanze) della classe attraverso le *variabili istanza*;
- La definizione della struttura dei dati comuni ad ogni classe attraverso le *variabili classe*.

Queste strutture vengono definite con:

- Variabili possedute dalla classe (**campi**);
- Codice che manipola gli oggetti, o in particolare i membri di classe (**metodi** e non *funzioni*);
- I meccanismi per la costruzione di oggetti (**costruttori**).

Variabili e funzioni di classe (cioè campi e metodi) sono anche detti **membri**.

Una classe in Java si istanzia sempre con la parola chiave `new`, in quanto non si possono allocare classi Java sullo stack, solo sull'heap. Abbiamo quindi che:

```
Classe cl1;
```

non istanzia alcuna classe, ma esiste solo come un oggetto *riferimento* a oggetti di tipo `Classe`, che colleghiamo ad un oggetto vero e proprio sull'heap come:

```
Classe cl1 = new Classe(/* argomenti del costruttore */);
```

Il fatto che gli oggetti dichiarati come `Classe` sono solo riferimenti ad oggetti sull'heap significa che facendo cose come `cl1 = cl2` stiamo solo spostando il riferimento da un oggetto all'altro, e non copiando le variabili istanza dei singoli oggetti.

Per accedere alle variabili istanza usiamo la classica notazione a punto (`Classe.variabile`, ecc...). Nessuno ci nega di includere riferimenti ad altre classi come membri di classi, dobbiamo solo ricordare che stiamo parlando sempre di riferimenti e non di oggetti classe veri e propri.

1.4.1 Costruttori

I costruttori sono sequenze di istruzioni utilizzate per definire lo stato iniziale di un oggetto in fase di instanziazione (quando viene usata la `new`), nel caso questo sia necessario, cioè i meccanismi standard non sono sufficienti (costruttore di default nullo) o ci sono informazioni note solo al momento della `new`.

I costruttori del Java:

- Hanno lo stesso nome della classe;
- Non restituiscono valore;
- Possono essere eseguiti solo in occasione della `new`;
- Possono avere 0 o più argomenti.