Appunti Programmazione Avanzata

Luca Seggiani
2025

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

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*.

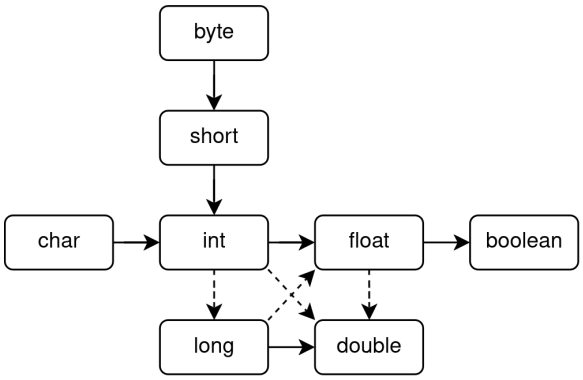
Vediamo poi che la parola chiave `const` non è implementata in Java (anche se è comunque riservata), mentre sono disponibili le parole chiave `static` e `final`. In particolare questa assegna a variabili valori finali, cioè che non possono più essere alterati.

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

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:

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 (secondo il 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.

2 Lezione del 25-09-25

Continuiamo a vedere gli aspetti del Java legati alle classi.

2.1 Metodi

Java supporta i **metodi** (funzioni definiti all'interno di classi) *statici* e *non statici*. Per questi metodi è supportato l'*overloading*, cioè lo stesso nome finche gli argomenti (cioè complessivamente la *firma*) variano.

Ogni volta che un metodo viene chiamato sullo stack viene creato un *record di attivazione*, cioè una struttura dati che contiene:

- Lo spazio riservato per gli argomenti formali, inizializzati al valore passato negli argomenti di chiamata;
- Lo spazio riservato alle variabili locali del metodo;

Al termine dell'esecuzione del metodo il record di attivazione corrispondente viene distrutto. Chiaramente la natura dello stack implica che i record verranno naturalmente creati e distrutti seguendo l'innestamento delle chiamate di metodo.

2.1.1 Passaggio di parametri

Il passaggio dei parametri ai metodi è *sempre* **per valore**, cioè nuove variabili vengono create col valore passato, e la variabile originale non viene modificata.

Notiamo che nel caso di classi, per valore viene passato il *riferimento* alla classe e non la classe stessa, cioè l'istanza di classe rimane unica e allocata sull'heap.

Ricapitolando, abbiamo quindi che:

- Il metodo chiamato non può modificare il valore di una variabile di tipo primitivo del chiamante;
- Il metodo chiamato non può modificare il valore di una variabile di tipo riferimento del chiamante;
- Di contro, il metodo chiamato *può* modificare il valore di una variabile oggetto di cui ha ricevuto il riferimento del chiamante (ovvero i riferimenti del chiamante e del metodo sono variabili diverse che puntano allo stesso oggetto, come i puntatori del C/C++).

2.1.2 Variabili locali

Le variabili locali, come già accennato, non hanno un valore di default e prima che venga assegnato un valore sono `undefined`. Usare una variabile locale prima che il suo valore sia ben definito comporta un errore.

Esistono casi in cui il compilatore non è abbastanza intelligente da capire che una variabile otterrà sicuramente un valore fra tutti i rami possibili di esecuzione. In tal caso basta inveire contro Oracle ed assegnare un valore qualsiasi alla variabile in fase di inizializzazione.

2.1.3 Riferimento `this`

Il Java supporta il riferimento (non propriamente *puntatore*) this alla classe corrente.

Nel caso si voglia semplicemente usare un campo della classe non è necessario usare `this`, in quanto questo è già nello scope di visibilità del metodo.

```
class Class {
  int a;

  void prolisso() {
    this.a++; // inutile! basta a++;
  }
}
```

`this` diventa allora utile quando gli argomenti formali *oscurano* i campi propri della classe:

```
class Class {
  int a;

  void oscura(int a) {
    this.a++; // il campo della classe
    a++; // l'argomento formale
  }
}
```

L'uso della sintassi `this.campo = campo` è estremamente comune quando si definiscono costruttori che prendono i campi della classe come argomenti:

```
class Studente {
  String nome;
  String cognome;
  String matricola;

  Studente(String nome, String cognome, String matricola) {
    this.nome = nome;
    this.cognome = cognome;
    this.matricola = matricola;
  }
}
```

Una forma più compatta della stessa cosa è la seguente:

```
class Studente {
  String nome;
  String cognome;
  String matricola;

  Studente(String nome, String cognome, String matricola) {
    this(nome, cognome, matricola);
```

```
  }
}
```

Un'altro caso di utilizzo di `this` è quando abbiamo effettivamente bisogno di un riferimento esplicito alla classe d'appartenenza del metodo (si pensi a un metodo che iscrive la sua istanza di classe ad una lista):

```
class Class {
  void iscriviti(Class[] lista, int idx) {
    lista[idx] = this;
  }
}
```

2.2 Membri statici

I campi e metodi statici sono utili a gestire informazioni relative all'intera classe e non agli oggetti istanza di classe generati da questa.

L'esempio tipico dei membri statici è per avere comportamento simile a *"fabbriche"*:

```
class Veicolo {
  String propretario;
  static int contatore = 0; // i veicoli istanziati da questa classe

  Veicolo(String proprietario) {
    this.proprietario = proprietario;
    contatore++;
  }
}
```

I valori statici non vengono allocati sull'heap assieme al resto delle istanze di classe, ma nella memoria statica (in quanto esistono una volta sola per tutta la classe).

L'accesso ai membri statici si fa usando la notazione puntuale direttamente sul nome di classe. Si può usare il riferimento di un'istanza di classe per fare la stessa cosa, ma questo è sconsigliato in quanto poco chiaro nel suo scopo (un programmatore che guarda l'accesso al membro non può sapere se questo è statico o meno senza guardare all'interno della classe).

Usare classi di utilità che definscono solo metodi statici è un pattern comune che sostituisce quello che in altri linguaggi sarebbe effettivamente rappresentato dalle funzioni globali:

```
class Math {
  public static void Sqrt(float num) {
    // ...
  }

  // ...
}
```

Ricordiamo che lo stesso metodo `main()` è dichiarato come `public` e `static`.

Se dentro una classe si vuole usare un metodo statico di quella classe, si può chiaramente usare il nome del metodo senza ulteriori qualificatori.

I campi statici possono essere inizializzati all'interno della dichiarazione di classe, prendendo come valori iniziali letterali, altri valori statici o risultati di metodi statici. Chiaramente non si possono usare variabili istanza o risultati di metodi istanza (possibilmente sconosciuti al tempo di definizione della variabile statica).

Chiaramente, nei metodi statici non possiamo usare il riferimento `this`, in quanto non c'è nessuna istanza di classe a cui riferirsi. Di contro, possiamo usare:

- Membri statici;
- Argomenti statici o non statici.

2.3 Pacchetti

I **pacchetti** o *package* servono a:

- Raggruppare classi tra loro correlate;
- Evitare conflitti fra i nomi di classe;
- Organizzare i progetti in maniera modulare.

La convenzione del Java è quella di usare un file per ogni classe, col nome della classe. Usando la direttiva `package` possiamo inserire una data classe all'interno di un pacchetto (buona pratica è rispecchiare questa struttura con le directory del progetto):

```
// ClassA.java
package MioPacchetto;

class ClassA {
  // ...
}

// ClassB.java
package MioPacchetto;

class ClassB {
  // ...
}
```

In questo caso per riferirsi alle classi `ClassA` e `ClassB` qualificheremo con la notazione puntale il pacchetto:

```
MioPacchetto.ClasseA;
MioPacchetto.ClasseB;
```

Pratica piuttosto tipico è includere la notazione puntuale anche nei pacchetti, cioè specificare i nomi di pacchetto in maniera gerarchica (`categoria.pacchetto.sottopacchetto` , ecc...). Questo è estremamente utile se usato assieme alla pratica accennata di prima di rispecchiare la struttura dei pacchetti in quella delle directory, sopratutto in progetti particolarmente complessi con molte componenti.

Una convenzione tipica nell'industria per trovare nomi univoci di pacchetto è quella di usare nomi di dominio al contrario, ad esempio:

```
com.ibm.db. // ...
it.unipi.email // ...
```

questo va effettivamente di pari passo con la qualificazione gerarchica data dai nomi di dominio DNS, ricordando il fatto che nel loro contesto originale vengono messi al contrario.

2.3.1 Pacchetto anonimo

Le classi che non vengono messe esplicitamente in un pacchetto finiscono nel pacchetto senza nome di default. Questo è sconsigliabile in quanto la classe risulterà a questo punto inaccessibile a classi che appartengono a pacchetti con nome, cioè l'accesso ai pacchetti va dall'alto verso il basso e non viceversa (non si può usare l'operatore di risoluzione vuoto come in C/C++).

2.3.2 Pacchetti e librerie

Tutte le librerie Java vengono implementate come pacchetti. Di base, il JDK definisce alcune librerie standard fra cui:

- `java.lang`: contiene alcuni tipi di default del linguaggio;
- `java.io`: contiene strumenti per la gestione dell I/O;
- `java.util`: contiene alcune utilità;
- ecc...

2.3.3 Regole di accesso fra pacchetti

Non sempre una classe in un pacchetto può accedere a classi di altri o del solito pacchetto: le regole di accesso vengono definite dalle qualificazioni della classe.

Una classe *top-level* (cioè definita in un file di classe) può avere 2 possibili qualificazioni:

- `public`;
- non `public` (nessuna qualificazione).

Le classi `public` sono disponibili a classi dello stesso pacchetto o di altri pacchetti, le classi non `public` sono invece disponibili solamente a classi dello stesso pacchetto.

Pssiamo usare i nomi non qualificati per accedere a classi nel pacchetto in cui ci troviamo. Per accedere a classi in altri pacchetti dobbiamo invece usare il nome completamente qualificato.

Un'alternativa può essere quella di **importare** la classe o l'intero pacchetto (in questo caso si includono tutte le classi del pacchetto) usando la direttiva `import` all'inizio del file che definisce la classe corrente.

Nella direttiva import si può usare anche la *wildcard* `*` per importare tutte le classi del pacchetto. Questo rende più esplicito quali pacchetti stiamo importando: con la wildcard si importa solo il pacchetto specificato, senza anche tutti i sottopacchetti.

Siano ad esempio due pacchetti `abc` e `abc.def`, dove il secondo è sottopacchetto del primo. In questo caso potremo dire:

```
1 import abc; // importa anche abc.def
2 import abc.*; // non importa abc.def
3 import abc.def.*; // ok
```

Un pacchetto particolare è `java.lang`, che viene importato sempre in ogni file.

Nel caso 2 o più pacchetti importati contengano classi con lo stesso nome, occorrerà nuovamente usare i nomi completamente qualificati. Questo talvolta può accadere anche con le librerie di default (ad esempio sia `java.util` che `java.sql` definiscono una classe `Date`).

2.3.4 Classi, pacchetti e gerarchia di file

Abbiamo visto come la gerarchia dei file che definiscono classi nelle directory possono rispecchiare la struttura innestata dei nomi di pacchetto, e anzi che questa è prassi consigliata. Tale convenzione permette infatti al compilatore (e all'IDE che scegliamo di usare) di gestire in maniera efficiente la compilazione delle classi.

Vediamo pero che le regole che ci siamo dati finora (una classe per file) sono leggermente stringenti: la prassi effettiva è di definire *al più* una classe di tipo `public` all'interno di un file, e assicurarsi che questa classe dia il nome al file. A questo punto potremmo definire altre classi non `public` all'interno dello stesso file, sempre assicurandosi di seguire la buona pratica di mantenere negli stessi file classi semanticamente legate fra di loro (classi utilità a classi top level, ecc...).

Ritorniamo all'aspetto della compilazione: il compilatore, come abbiamo detto, sfrutterà la gerarchia di directory per trovare le classi importate attraverso il loro nome completamente qualificato. L'insieme di cartelle a partire dalle quali il compilatore (e la JVM, si parla di linking dinamico) effettua questa ricerca viene detto **classpath**.

Se non specificato il classpath è automaticamente preso alla directory corrente. Altrimenti, questo può essere specificato con l'opzione `-cp` o `-classpath` del comando `java`. Più punti di partenza possono essere specificati col separatore `:` in ambiente Unix.

In ogni caso, le classi di sistema fornite assieme alla JVM vengono trovate automaticamente.

2.4 Qualificatori di accesso

Abbiamo visto il qualificatore `public` (e di conseguenza il non `public`) per le classi top-level. Vediamone gli altri:

- `private`: il campo o il metodo può essere acceduto solo dalla stessa classe in cui è definito;
- Nessun qualificatore: già visto, il campo o il metodo può essere ecceduto dalla stessa classe in cui è definito e da tutte le classi dello stesso package;
- `protected`: il campo o metodo può essere acceduto dalla stessa classe in cui è definito e da tutte le sue sottoclassi (anche in altri package), nonché da tutte le classi nello stesso package;
- `public`: il campo o metodo può sempre essere acceduto.

2.5 Singoletti

La struttura delle classi del Java ci permettono di implementare **pattern** di classi. Uno di questi è il **singoletto**, cioè una classe di cui esiste una sola istanza in tutto il programma.

Questo si presenta più o meno come:

```
public class Singleton {
  private static Singleton m;
  private Singleton() { /* ... */ }; // definiamo costruttore privato

  public static void getInstance() {
    if(m == null) {
      m = new Singleton();
    }
    return m;
  }
}
```

In questo modo impediamo ad altri metodi di definire nuove istanze della classe `Singleton`:

```
Singleton more = new Singleton(); // errore! costruttore privato
```

2.6 Array

Vediamo come funzionano gli array in Java. Questi sono simili a quanto visto in altri linguaggi: contenitori contigui di elementi dello stesso tipo identificati da un indice. La loro dimensione è definita a tempo di esecuzione, cioè:

```
int[] a, // inizializzato un riferimento ma non esiste array
a = new int[5]; // adesso esiste array

// si puo' fare in una volta sola:
int[] b = new int[5];
```

Gli array sono gestiti quindi come le classi, cioè sono allocati nell'heap e li gestiamo attraverso riferimenti sullo stack.

Notiamo che, sebbene la dimensione sia determinata a tempo di esecuzione, la dimensione da lì in poi è fissa: per ridimensionare array bisogna usare le stesse tecniche per gli array del C/C++. Una funzionalità che non ci portiamo invece dal C++ è l'aritmetica dei puntatori, cioè preso un riferimento ad array non possiamo incrementare ma solo selezionare elementi dell'array:

```
int[] a = new int[5];
a[1]; // ok
a++; // errore
```

Nel caso di accessi fuori bound, l'accesso non è permesso e viene lanciata un'eccezione.

2.6.1 Valori di default in array

Per gli array esistono valori di default:

- `false` per i tipi booleani;
- 0 per i tipi numerici;
- `null` per i riferimenti.

Valori espliciti possono essere forniti con la notazione graffa, ed entrambe le forme sotto sono valide

```
int[] a = {2, 3, 4};
int[] a = new int[] {2, 3, 4};
```

2.6.2 Array di riferimenti

Se si creano array di riferimenti, al momento dell'inizializzazione dell'array non esistono oggetti, cioè:

```
Studente[] s1 = new Studente[5];
// esistono 5 riferimenti a studente ma nessuno studente!
s1[0].getMedia(); // errore
```

Ciò che possiamo fare è quindi assegnare ai riferimenti creati nuovi oggetti:

```
s1[0] = new Studente("Mario", "Rossi");
```

Esistono diverse forme di inizializzazione equivalenti:

```
Studente[] s2 = { new Studente("Mario", "Rossi"), new Studente("Luigi", "
    Verdi") };
Studente[] s3 = new Studente[] { new Studente("Mario", "Rossi"), new
    Studente("Luigi", "Verdi") };
```

2.6.3 Array multidimensionali

Per creare un'array multidimensionale si usa la solita sintassi a parentesi quadre multiple:

```
Cell[][] grid = new Cell[HEIGHT][WIDTH];

grid[y][x] = Cell.grass;
```

La struttura generata sarà la classica configurazione a direttorio degli array multidimensionali, dove indirizziamo array con array da sinstra verso destra.

L'ultima dimensione può essere omessa, cioè si può creare un direttorio di dimensione fissa che riferisce ad array di dimensione variabile:

```
int[][] grid = new int[3][]; // ok
int[][] grid = new int[][3]; // errore
```

Si possono ottenere array multidimensionali non rettangolari, ad esempio come:

```
int[][] triangle = new int[3][];

for(int i = 0; i < 3; i++) {
  triangle[i] = new int[i + 1];
}
```

2.6.4 Metodi su array

Esistono metodi di utilità contenuti nella classe `System` pensati per gestire le Array. Questi sono ad esempio:

- `arrayCopy(Object src, int srcPos, Object dest, int destPos, int length)` per la ricopiatura da regioni di array ad altre regioni di array. Secondo la logica discussa in 2.3, dovremo chiamare questa funzione come `System.arrayCopy();`
- ecc...

Esiste anche la classe `Arrays` che espone alcuni metodi di utilità specifici alle array:

- `sort(Object array)` per l'ordinamento di array;
- ecc...

Ulteriori metodi possono essere trovati nella documentazione di java, a `docs.oracle.com/javase/<versione>/docs/api/`.

Inoltre, notiamo che la lunghezza di un'array è accessibile come proprietà usando `.length`.

2.7 Stringhe

Le stringhe in Java non sono le stringhe terminate del C/C++, ma istanze di classe `String`. Le stringhe definite come istanze di questa classe sono **immutabili**: per ottenere stringhe mutabili dobbiamo usare le classi:

- `StringBuffer`: può essere manipolata da più *thread* (è *thread-safe*);
- `StringBuilder`: deve essere manipulata da un solo thread.

Le stringhe, essendo istanze di classe, vengono puntate da riferimenti ed allocate sull'heap.

Si possono concatenare con l'operatore `+`, che può essere anche applicato a stringhe e valori di altro tipo (provocando una conversione implicita dal tipo usato a stringa).

2.7.1 Metodi su stringhe

Come per le array, disponiamo di alcuni metodi utili sulle stringhe:

- `substring(int beginIndex, int endIndex)`: restituisce la sottostringa dall'indice `beginIndex` all'indice `endIndex`;
- `.length()`, definito come metodo istanza e non come proprieta (come era stato per le array), restituisce la lunghezza della stringa;
- ecc...

Possiamo costruire stringhe a partire da array di caratteri possiamo usare la sintassi:

```
char[] chars = {'h', 'e', 'l', 'l', 'o'};
String str = new String(chars);
```

Per effettuare il confronto fra stringhe dobbiamo usare il metodo `equals()`, e non l'operatore `==`, in quanto questo confronta i riferimenti e non i valori delle stringhe riferite.

3 Lezione del 30-09-25

3.1 Metodo main

L'entrypoint di un programma in Java è il metodo `main()` (dichiarato `public` e `static`), definito all'interno della classe `Main` (dichiarata `public`).

`main()` ha solitamente come argomenti l'array di riferimenti a `String` detta `args`, che contiene gli argomenti con cui è stato chiamato il programma.

L'esempio tipico sarà quindi:

```
public class Main() {
  public static void main(String[] args) {
    // entrypoint del programma
  }
}
```

Si possono avere più di un metodo `main()` finché se ne definisce uno per ogni classe (ergo possono esistere classi non `Main` che possiedono un metodo `main()`).

Questo può essere utile a elaborare (in maniera non proprio strutturata) test intermedi sulle classi che si implementano.

Un altro caso utile può essere nel caso di applicazioni che implementano più modalità di esecuzione (magari GUI e CLI), dove ogni `main()` corrisponde ad una certa modalità.

3.2 Ereditarietà

Il Java, essendo un linguagio OOP, implementa l'**ereditarietà** delle classi: si può definire una classe come *figlia* di un altra, e in questo caso ne *eredità* campi e metodi.

Per fare questo si usa la parola chiave `extends`:

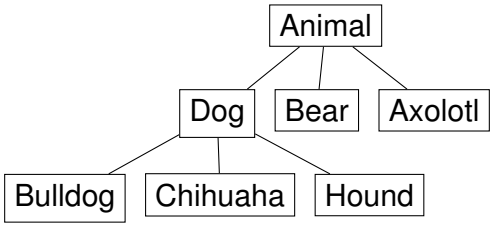
```
class Animal {
  // implementazione di Animal
}

class Dog extends Animal {
  // implementazione di Dog
}
```

Una limitazione è che una classe figlia può avere una sola classe padre (a differenza di linguaggi come il C++). Vedremo che per replicare la solita funzionalità si usano le *interfacce*.

La relazione che viene stabilita quando si usa il meccanismo dell'ereditarietà è quella dell'*"è un"*, cioè con riferimento all'esempio sopra, un `Dog` *è un* `Animal`.

La definizione della classe figlia verrà quindi fatta in base alle *differenze* con la classe padre, cioè ridefinendo i campi del padre o aggiungendone altri. I campi privati vengono ereditati, ma la classe figlia non può accedervi.

Per rappresentare le gerarchie di classi graficamente si usano diagrammi di gerarchia come il seguente:



dove la relazione *"è un"* si svolge dall'alto verso il basso.

Uno dei chiari vantaggi dell'uso dell'erediterietà è che un riferimento alla classe padre può riferire a istanze della classe figlia. Questa si dice conversione da sottotipo a supertipo. Si può effettuare anche la conversione opposta: questo chiaramente implica che la conversione sia valida (cioè la superclasse sia specializzata in una classe del tipo che cerchiamo).

Per valutare la validità della conversione è presente l'operatore `instanceof`, che restituisce `true` quando si confronta un'istanza di classe col suo tipo.

La regola generale è quindi che non conta il tipo del riferimento, ma il tipo dell'oggetto riferito.

Nel caso di chiamate a metodi di oggetti riferiti, il tipo del riferimento determina solo l'*insieme* dei metodi che possono essere invocati. L'implementazione effettiva del metodo viene definita dal tipo reale della classe, cioè chiamando metodi specializzati in classe figlie attraverso riferimenti a classi padri, si chiamano comunque i metodi specializzati e non quelli delle classe padri.

3.2.1 Riferimento `super`

All'interno di classi figlie possiamo usare la parola chiave `super` per riferirci alla classe padre. Questo può essere utile quando si ridefiniscono metodi e si vuole chiamare la versione del metodo nella classe padre (magari per poi aggiungere altra funzionalità).

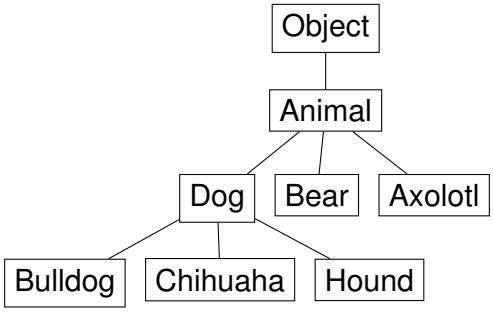
La parole chiave `super` può essere usata anche per chiamare il costruttore della classe padre (`super()`). In questo modo possiamo inizializzare la sezione di membri della classe padre quando si costruisce la classe figlia.

Se la classe padre è dotata di soli costruttori con argomenti, chiamare questo costruttore è obbligatorio.

3.2.2 Classe `Object`

Esiste una classe di sistema definita in `java.lang`, chiamata `Object`. Se una classe non indica esplicitamente di estenderne un altra, allora è implicitamente figlia della classe `Object`.

La gerarchia reale avrà quindi questo aspetto:



`Object` ha alcuni metodi, che possiamo dividere in 2 categorie:

- Metodi per la sincronizzazione fra thread (`wait()`, `notify()`, `notifyAll()`);
- Metodi di utilità generale (`equals()`, `hashcode()`, `toString()`)

3.2.3 Metodo `equals()`

Il metodo `equals()` ha la firma:

```
1 public boolean equals(Object o)
```

cioè prende come argomento un oggetto è confronta l'oggetto implicito (quello su cui è chiamato) con tale oggetto. A questo punto, restituisce `true` se i 2 oggetti risultano uguali, e `false` altrimenti.

Ogni classe può ridefinire il metodo `equals()` secondo il proprio criterio.

Per esempio, la classe `String` ridefinisce il metodo in modo tale da verificare se due stringhe sono uguali (composte dagli stessi caratteri).

L'implementazione di default della classe `Object` equivale invece all'operatore `==` (cioè per i riferimenti controlla se riferiscono lo stesso oggetto).

3.2.4 Metodo `getClass()`

Il metodo `getClass()` è definito per ogni oggetto `Object` e ci dà il primo sguardo al meccanismo della **riflessività**. La riflessività è il modo con cui il Java ci permette di guardare al codice sorgente del programma che stiamo sviluppando.

Il metodo `getClass()` restituisce quindi un oggetto di tipo `Class` (con la C maiuscola) che descrive la classe stessa dell'oggetto su cui è chiamata.

Possiamo usare questo metodo per capire se due classi hanno lo stesso tipo, o se una classe ha un dato tipo confrontandolo con i cosiddetti *letterali classe*, definiti come `<classe-nota>.class`.

3.2.5 Metodo `hashCode()`

Un'altro metodo definito per `Object` è `hashCode()`, che restituisce l'`hashCode` dell'oggetto a cui viene applicato. Questo è un valore che deve essere diverso per ogni oggetto (solitamente associato a dove l'oggetto è allocato).

Dobbiamo quindi impegnarci a ridefinirlo (nel caso ci torni utile) con campi che sono unici per ogni istanza di classe.

3.2.6 Metodo `toString()`

Infine, vediamo il metodo `toString()`. Questo serve semplicemente a convertire un oggetto in una stringa, e può essere ridefinito nella maniera che ha più senso (semanticamente) per il tipo di oggetto che stiamo implementando.

3.3 Polimorfismo

Il principio dell'ereditarietà, e principalmente il fatto che un riferimento di tipo superclasse può riferire ad oggetti ti tipo sottoclasse, ci permette di realizzare un'altro principio, quello del **polimorfismo**.

Supponiamo infatti che una superclasse implementi un metodo, e che più sottoclassi eredi di tale classe ridefiniscano dato metodo. Ottenendo un riferimento dei tipo superclasse a un insieme di classi figlie, potremmo chiamare il metodo ridefinito (dal riferimento di tipo superclasse) sulle sottoclassi, ottenendo ogni volta il comportamento ridefinito dalla sottoclasse.

3.4 Classi astratte

Una **classe astratta** non può essere istanziata, ma può essere usata come classe base nella definizione di nuove classi. Inoltre, ci è concesso creare riferimenti a classi astratte che possono riferire a ogni tipo specializzato dalla stessa.

Le classi astratte sono quindi utili come modello per la definizione di classi: ci permettono di definire campi e metodi (nel caso non si specifichi corpo, *metodi astratti*) comuni a più classi figlie ma che di per sè non corrispondono a nessuna implementazione completa di classe.

4 Lezione del 07-10-25

4.1 Classi final

All'opposto delle classi astratte ci sono le classi `final`: queste sono classi che non possono essere estese. I metodi definiti nelle classi `final` sono necessariamente `final`, cioè non sovrascrivibili.

In Java, le classi `final` sono poche: principalmente si usano metodi `final` per impedire a chi estende la classe che contiene il metodo di modificarne il comportamento.

4.2 Interfacce

Le **interfacce** sono un paradigma simile a quello delle classi astratte, ma solitamente più usate in Java.

Dal punto di vista sintattico si definiscono come una raccolta di metodi astratti: in pratica ci permettono di definire modelli di comportamento standard che più classi possono implementare. Quando una classe implementa un interfaccia, chi sa interagire con quell'interfaccia sa automaticamente interagire con quella classe. In particolare, si possono creare *riferimenti* di tipo interfaccia, ma non *oggetti* di tipo interfaccia. I riferimenti di tipo interfaccia riferiranno a un qualsiasi oggetto che implementa l'interfaccia.

Per definire un'interfaccia si usa la parola chiave `interface` e una sintassi simile a quella delle classi. Automaticamente tutti i metodi definiti sono `abstract` e `public`.

4.2.1 Implementare interfacce

Per una classe, implementare l'interfaccia (attraverso la parola chiave `implements`) significa definire *tutti* i metodi dell'interfaccia, a meno che la classe sia astratta. Una classe può implementare tutte le interfacce che vuole: solo le `extends` sono forzatamente unarie. Si può quindi avere.

```
class A extends B implements C, D, ... { /* ... */ }
```

4.3 Tipi enumerazione

I **tipi enumerazione** sono sostanzialmente interfacce che definiscono variabili costante intere (i campi sono automaticamente `public`, `static` e `final` nelle interfacce), che si possono poi usare come identificatori costanti (e semanticamente coerenti con lo scopo del tipo enumerazione).

Si può usare la parola chiave `enum` per non dover definire esplicitamente gli interi, ad esempio:

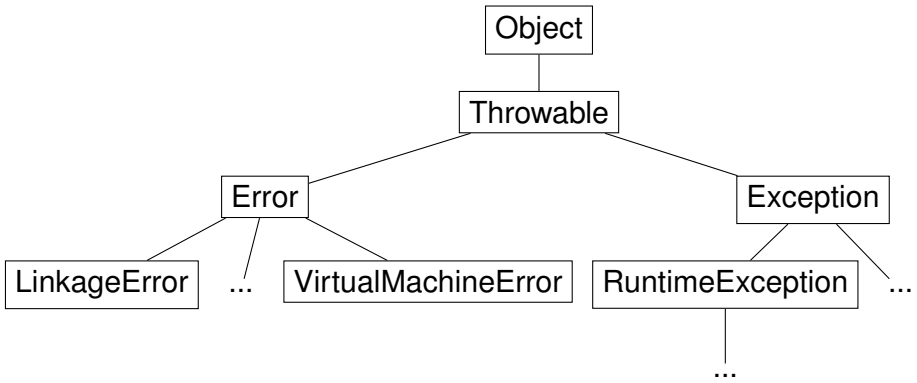
```
enum Colori {
  ROSSO,
  VERDE,
  GIALLO
}
```

Abbiamo che gli `enum` in Java sono effettivamente definiti come una classe, la classe `Enum<E>` implementata in `java.lang.Enum`.

Sono per questo dotati di alcune funzioni di utilità, tra cui `toString()` e `name()` per ottenere i nomi dei campi dell'`enum` a tempo di esecuzione (impossibile in linguaggi come il C++, a meno di non definire macchinose conversioni esplicite in stringhe). In particolare, la differenza fra `toString()` e `name()` è che `name()` è `final`, mentre `toString()` è ridefinibile da ogni classe `enum`.

4.4 Eccezioni

Il Java permette la gestione delle eccezioni secondo il meccanismo noto ad esempio dal C++. In particolare, la tassonomia delle classe di eccezione è la seguente:



L'oggetto Throwable può essere *lanciato*, cioè mandato al chiamante attraverso la parola chiave `throw`. Questo definirà il classico blocco `try { /* ... */} catch(Exception e) { /* gestore */ }` per implementare il gestore di eccezioni.

Esistono eccezioni di tipo **checked** e **unchecked**:

- **Checked**: fanno parte della *firma* della funzione che le lancia, e bisogna quindi che queste lo dichiarino (con la parola chiave `throws`) e il chiamante definisca un gestore (o a sua volta dichiari di poter inoltrare l'eccezione). Sono di questo tipo tutte le classi che estendono Exception tranne RuntimeException;

- **Unhecked**: rappresentano errori di programmazione o comunque non rimediabili a tempo di esecuzione, e non è necessario gestirne un gestore (ci aspettiamo che sia il gestore di default a stampare informazioni ed arrestare il programma). Sono di questo tipo le specializzazioni di:
  - **Error**: rappresentano errori interni alla JVM, o comunque specifici del linguaggio Java. Non ci aspettiamo che il progammatore possa gestirli, e quindi terminano il programma;
  - **RuntimeException**: rappresentano errori di programmazione irrecuperabili come accessi fuori buond ad array, ecc...

4.4.1 Blocchi `try-catch-finally`

Vediamo nello specifico come si trattano le chiamate a metodi che possono lanciare eccezioni, e il modo in cui si definiscono gestori di eccezioni. Un classico blocco `try-catch-finally` in Java ha l'aspetto:

```
try {
  // statement
} catch(Exception1 e1) {
  // gestisci Exception1
} catch(Exception2 e2) {
  // gestisci Exception2
}
// ...
finally {
  // blocco finally
}
```

Vediamone le componenti:

- Il blocco `try` contiene il codice che vogliamo eseguire, cioè probabilmente una chiamata a metodo che lancia Exception1 e Exception2;
- Il blocco `catch` cattura una certa eccezione (specificata negli argomenti, può catturare più eccezioni se si usa il separatore `|`). A questo punto dovrà definire codice che si occupa di gestire tale eccezione;
- Il blocco `finally` contiene codice che viene eseguito dopo che tutta l'elaborazione precedente, sia stata dalla `try` o dalle `catch`, è stata eseguita. Ha principalmente lo scopo di permetterci di fare pulizia se un'operazione su risorse (come un accesso a file) fallisce.

I blocchi `catch` vengono valutati sequenzialmente quando viene lanciata un eccezione: è inutile definire un gestore di eccezione se in seguito se ne definisce uno per un suo sottotipo, in quanto il primo verrà sempre eseguito al posto del secondo quando possibile (e in questo caso è possibile sempre).

4.4.2 Clausola `throws`

Abbiamo introdotto come la parola chiave `throws` specifica per il *contratto* della funzione che questa può lanciare una certa eccezione *checked*: approfondiamo questo aspetto. Inserire le eccezioni lanciate costringe il programmatore che usa il metodo a capire cosa potrebbe succedere, e risolvere correttamente le situazioni critiche. Proprio per questo

motivo, a dispetto del polimorfismo è opportuno che le eccezioni specificate siano il più specifiche possibile.

I blocchi di inizializzazione statici non possono lanciare eccezioni *checked* in quanto non possiedono clausola `throws`: di contro i blocchi di inizializzazione non statici prendono direttamente la clausola `throws` del costruttore di classi (e infatti ne sono considerati parte).

Nel caso dell'ereditarietà, abbiamo che:

- Se si sovrascrive un metodo con clausola `throws`, bisogna specificare una causola `throws` del metoodo sovrascritto con la stessa eccezione o un suo sottotipo (a meno che il metodo sovrascritto non lanci eccezioni di qualsiasi tipo). Sostanzialmente, vogliamo assicurarci che chiunque vede soltanto il metodo base (ad esempio da un riferimento a superclasse) possa gestire l'eccezione;
- Se un metodo esiste in più interfacce, e ha clausole `throws` diverse in ognuna, dovrà implementarle tutte nella classe che implementa le interfacce.

5 Lezione del 14-10-25

5.1 Asserzioni

Le **asserzioni** in Java permettono di ineserire condizioni presunte vere (utili al debug) da verificare a tempo di esecuzione. La sintassi è `assert <espressione>`, dove l'espressione deve essere booleana e valutare a *vera*.

Il fallimento di un'asserzione genera un errore di tipo *AssertionError*.

Un'altra possibile sintassi per le asserzioni è `assert <condizione1> : <condizione2>`, dove la condizione 1 è analoga alla precedente, mentre la condizione 2 è una stringa viene valutata se la prima vale falso per riportare informazione di diagnistica.

Ad esempio, si potrà dire:

```
1 assert x == y : "x vale" + x " e y vale " + y;
```

Il blocco incodizionato del codice (magari se si salta ad un frammento di codice che dovrebbe essere irraggiungibile) si fa semplicemente con `assert false`.

Di norma, la JVM (`java ...`) esegue il codice con le asserzioni disabilitate. Se si vogliono abilitare, bisogna fornire l'argomento `-enableassertions` (abbreviato `-ea`).

5.2 Thread

I **thread** in Java sono un astrazione per i flussi di esecuzioni indipendenti.

I thread sono flussi diversi che vivono all'interno dello stesso *processo*: ergo hanno accesso alle stesse risorse e allo stesso spazio di indirizzamento. In questo possono passarsi oggetti fra di loro senza doversi appellare ai servizi di **IPC** (*Inter Process Communication*) forniti dal S/O.

Un esempio di utilizzo di thread può essere in un'applicazione server, dove ad ogni client che effettua richieste si associa (finché possibile) un suo thread dedicato. Questo semplifica sia lo sviluppo lato server (un singolo thread gestisce un singolo client), che l'esperienza lato client (si andrà a parlare con un singolo thread).

5.2.1 Thread e multithreading

Ricordiamo che i thread sono un'astrazione. Nei moderni processori, sappiamo di aver a disposizione più di un unità di elaborazione (*core*), per cui l'esecuzione parallela di istruzioni è effettivamente possibile. Questo non si traduce direttamente nell'esecuzione di ognuno dei nostri thread in Java su un core diverso: è compito del S/O capire se delegare a più core l'esecuzione dei thread, o se eseguirli tutti su un solo core in *time-slicing*.

5.2.2 Thread default

Quando mandiamo in esecuzione un programma Java, la JVM crea un thread detto *main thread* (cioè il thread *principale* o *di default*) che esegue il codice definito nel metodo `main` della classe specificata.

Questo metodo può poi definire, attraverso il suo flusso di esecuzione, altri thread che si evolveranno quindi parallelamente a quello principale.

5.2.3 Oggetto thread

Il modo idiomatico in Java di realizzare la funzionalità dei thread è quello di sfruttare un apposito oggetto, l'oggetto *Thread*.

L'oggetto Thread definisce un metodo, `run()`, all'interno del quale possiamo definire il flusso di esecuzione proprio del thread.

6 Lezione del 16-10-25

Ritorniamo sull'argomento dei thread.

Aggiungiamo, rispetto alla scorsa lezione, che ogni thread ha il suo stack (con il suo record di attivazione). Inoltre, tutti i thread condividono lo stesso heap. Il GC non rimuove gli elementi che sono raggiungibili da qualunque thread.

6.0.1 Interfaccia Runnable

Visto che in Java l'ereditarietà è unitaria, dover sempre estendere la classe *Thread* può essere limitante. In questo Java fornisce l'interfaccia *Runnable*, che quando implementata rende una classe *eseguibile* (come una lambda o un funtore).

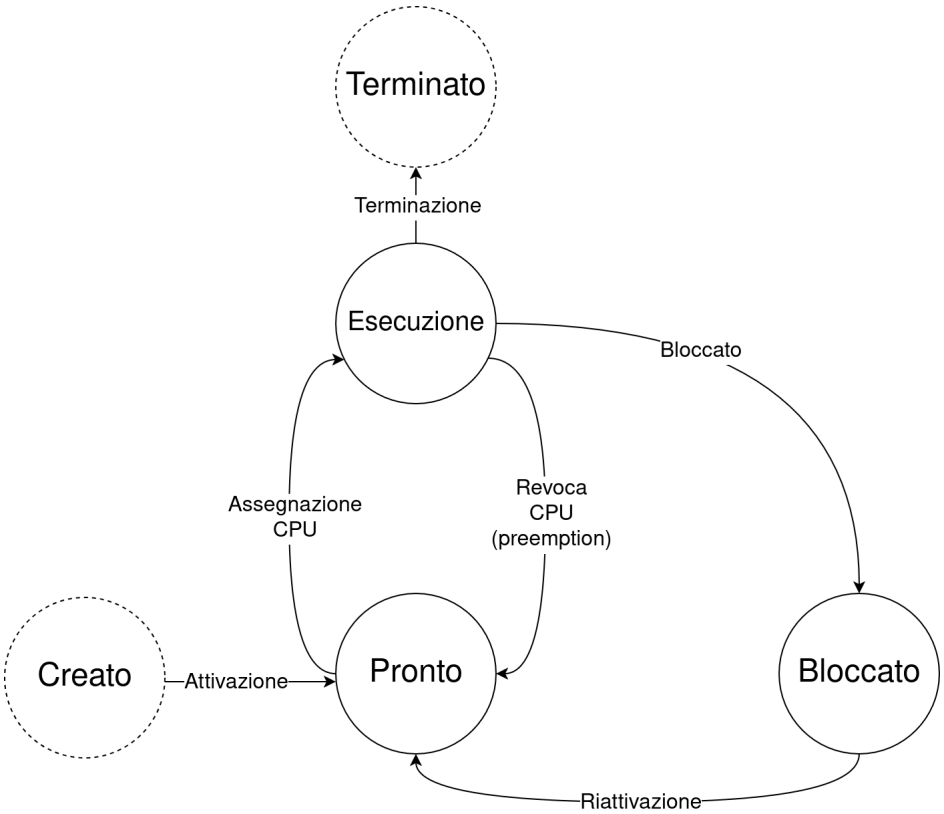
Come per la classe Thread, l'interfaccia Runnable è provvista di un metodo astratto `run()` da ridefinire per specificarne il comportamento.

Quando si crea un'oggetto *Thread* si può quindi usare un'oggetto che implementa l'interfaccia Runnable come argomento al costruttore:

```
class MyThread implements Runnable {
  @Override
  void run() {
    System.out.println("Sono un thread");
  }
}

// da qualche altra parte
Thread myThread = new Thread(new MyThread());
myThread.start();

// stampera' "Sono un thread" da un nuovo thread
```

6.1 Stati di un thread

Un thread può trovarsi in diversi stati, schematizzabili attraverso lo schema (preso direttamente dagli appunti di sistemi operativi):



In Java le transizioni fra stati possono corrispondere a particolari chiamate di metodi da dentro l'oggetto thread. In particolare:

- *Pronto*: il thread arriva qui quando si chiama il metodo `start()`, e può tornarci alla fine del quanto di tempo assegnatogli, per *preemption*, o per concessione volontaria (attraverso `Thread.yield()`).

  Inoltre, torna qui se era bloccato e viene sbloccato (fine `sleep()`, `notify()`, blocchi su risorse rilasciati, ecc...);

- *Bloccato*: un thread arriva qui quando effettua una sospensione volontaria (`sleep()`, `wait()`, blocchi `synchronized`) o quando viene bloccato involontariamente su risorse (file, I/O, ecc...);

- *Esecuzione*: questo è lo stato in cui il thread effettivamente esegue.

6.1.1 Thread e S/O

Il concetto di thread in Java è chiaramente parallelo a quello di thread nei S/O.

Vediamo una breve outline cronologica di questa relazione:

- Oggi la prassi è quella di associare ad ogni thread Java un thread di S/O, e lasciare che siano i meccanismi di gestione dei processi del S/O a gestire i thread Java in esecuzione;

- Storicamente, questo non è stato sempre il caso: le prime JVM erano infatti processi monolitici, con un unico flusso di esecuzione, che gestivano internamente tutti i diversi thread.

In ogni caso, le specifiche del linguaggio si dimostrano (volutamente) blande sull'argomento thread, in quanto la maggior parte delle problematiche associate alla gestione di questi è associata alla struttura specifica del S/O.

6.1.2 Priorità thread

Di un thread Java si può impostare la priorità (un numero da 1 a 10, via `setPriority (int prio)`), e questa dovrebbe grossomodo corrispondere alla priorità del corrispondente thread S/O (grossomodo in quanto l'S/O potrebbe prevedere più di 10 livelli di priorità).

Per questo motivo, bisognerebbe usare la priorità dei thread Java solo come indicazione per il S/O e come ottimizzazione, e non fidarsi completamente di ottenere del comportamento sperabile.

Il thread di default parte con priorità normale (il valore medio, cioè 5). Quando si creano nuovi thread, questi ereditano la priorità del thread padre (quindi 5 dal thread di default).

La schedulazione dei thread Java è in genere di tipo *fixed priority*.

6.1.3 Terminare e riattivare thread

Quando un thread termina, non si può usare nuovamente `start()` per rimetterlo in esecuzione, ma bisogna crearne una nuova istanza ed eseguire quella.

Se da un thread si vuole terminare l'esecuzione del solo thread, si può chiaramente ritornare da `run()`. Se si vuole invece terminare l'esecuzione dell'intero programma, si può usare `System.exit()`.

6.1.4 Metodo sleep

Il metodo `sleep(long t)` permette di sospendere il thread per il numero di millisecondi specificato in `t`.

Il metodo può lanciare l'eccezione *InterruptedException*, e questo accade quando il thread viene interrotto. Per questo bisogna definire un blocco `catch` che catturi tale eccezione e fornisca un qualche tipo di handler:

```
1  void run() {
2    while(true) {
3      System.out.println("Tick");
4      try {
5        Thread.sleep(1000);
6      } catch(InterruptedException e) {
7        System.err.println("Il thread e' stato svegliato. " + e.getMessage()
       );
8      }
9    }
10 }
```

6.2 Corse critiche

Un problema in cui si può incorrere quando si sfruttano i thread è quello delle **corse critiche**.

Quando più thread interagiscono con lo stesso oggetto, infatti, potrebbero lasciarlo in stati inconsistenti a causa di scritture contemporanee che si cancellano fra di loro. In altre parole, le operazioni su campi di oggetti in Java non sono (di default) *atomiche*, e i metodi che le implementano possono essere deschedulati (per preemption) quando l'oggetto su cui stanno lavorando è ancora in stato inconsistente.

6.2.1 Monitor, metodi `synchronized`

In Java questo problema viene risolto sfruttando il modello dei **monitor**. Secondo questo modello, ad ogni oggetto è associato un **lock**, che può essere *aperto* o *chiuso*.

Per agire su un oggetto, un thread deve acquisire un lock. Quando avrà finito la sua operazione, potrà rilasciare tale lock.

Dal punto di vista sintattico, questo si implementa usando il modificatore (applicabile ai metodi) `synchronized`. Quando un thread esegue un metodo `synchronized` e agisce su un'oggetto x, quello che accade è:

- Acquisisce il lock di x;
- Esegue il corpo del metodo ;
- Rilascia il lock di x.

Tutti i metodi dichiarati come `synchronized` condividono per ciascun oggetto un lock (il lock predefinito). I metodi non `synchronized` possono invece bypassare il meccanismo dei lock, anche quando un metodo che invece è `synchronized` ha il lock sull'oggetto cercato.

Per questo è importante dichiarare come `synchronized` tutti i metodi che rischiano di portare oggetti in stati inconsistenti. Di contro, è abbastanza naturale definire metodi che non richiedono di rispettare i lock (metodi non bloccanti, di lettura, ecc...).

Abbiamo quindi che i metodi `synchronied` *cercano* di ottenere i lock sugli oggetti che modificano. I lock si acquisiscono per conto del thread corrente: se quel thread è già in possesso del lock cercato, non si blocca e va direttamente all'esecuzione del metodo.

Nel caso di metodi `synchronized` il lock viene rilasciato:

- Al normale termine del esecuzione del metodo;
- In caso di situazioni particolari come le eccezioni.

6.2.2 Blocchi `synchronized`

La parola chiave `synchronized` può essere usata anche per definire *blocchi* di codice, e non solo metodi. In questo caso tali blocchi si comportano esattamente come i metodi che abbiamo descritto finora.

I blocchi sincronizzati accettano un argomento, che è quello su cui vogliamo bloccare. Nel caso di metodi sincronizzati, l'oggetto è implicitamente la classe del metodo:

```
1 class A {
2   void synchronized foo() {
3     // sincronizza su A
4   }
```

```
5
6     void bar() {
7       synchronized(this) {
8         // come sopra ma inutilmente esplicito
9       }
10    }
11  }
```

Coi blocchi sincronizzati si possono però avere altri oggetti su cui sincronizzare. Il loro uso è quindi necessario quando:

- Non tutto il codice di un metodo deve essere eseguito in mutua esclusione (sincronizzato). Questo è particolarmente utile aumentare il grado di parallelismo dei thread: se si sincronizza solo quando gli oggetti effettivamente ci servono, si lascia tempo agli altri thread per lavorare sullo stesso oggetto;
- Si vuole sincronizzare su oggetti diversi da quello implicito. Ad esempio, si potrebbe voler dire:

  ```
  1 void metodo(Parametro p) {
  2   synchronized(p) {
  3     // qui abbiamo il lock su p
  4   }
  5 }
  ```

  In questo caso l'oggetto di cui si prende il lock non è più la classe corrente, ma l'oggetto di tipo *Parametro* che passiamo nell'argomento *p*.

  Questo può essere utile quando si usano oggetti che non prevedono la sincronizzazione di default: in questo modo imponiamo *noi* che le operazioni che svolgiamo vengano sincronizzate;
- Si vuole sincronizzare su un array. Questo in quanto le array non possono essere soggetto di lock impliciti (non sono classi che definiscono metodi). La sintassi è in questo caso analoga all'esempio precedente.

Più lock su più oggetti possono essere ottenuti innestando blocchi `synchronized`.

6.2.3 Lock statici

Quando il modificatore `synchronized` viene usato su metodi statici il lock implicito non è sull'*istanza* di classe, ma sull'intera classe. In questo modo si possono definire metodi che esegono in maniera mutualmente esclusiva sull'intero oggetto classe.

6.3 Deadlock

Nel caso due thread debbano ottenere un lock appartenente all'altro per portare avanti la loro operazione, e non siano ancora in condizioni di rilasciare il loro lock, si incorre in una situazione detta **deadlock**.

Un risultato teorico è che *se si prendono i lock sempre nello stesso ordine*, i deadlock non possono verificarsi. Questo chiaramente non è sempre facile dal punto di vista implementativo.

6.4 Sincronizazzione

Veniamo a come eseguire operazioni in thread solo nella condizione che un altro thread abbia eseguito la sua operazione. Questo è un problema di *sincronizzazione* o in particolare di **comunicazione** fra thread.

Facciamo attenzione al fatto che la parola chiave `synchronized` è fuorviante: in verità questa risolve un problema di *interferenza*, o più propriamente di **mutua esclusione** dell'accesso a determinate risorse.

Il problema che vogliamo invece risolvere adesso è di *sincronizzazione* temporale nell'esecuzione di operazioni in stretto ordine cronologico.

In Java questo problema viene risolto sfruttando metodi definiti direttamente sulla classe *Object*:

- `notify()`: permette di *notificare*, appunto, un oggetto nel *wait-set* dell'oggetto che lo chiama (più dettagli in seguito);
- `notifyAll()`: come sopra, ma notifica tutti gli oggetti nel *wait-set*;
- `wait()`: permette di sospendere l'esecuzione di un thread fino al verificarsi di una determinata condizione (notificata da qualcun'altro). Questo è un metodo che non restituisce nulla ma può lanciare la *InterruptedException*. Quando si chiama `wait()` i lock dell'oggetto vengono rilasciati e acquisiti automaticamente al rientro in esecuzione.

  Ad ogni oggetto è associato un *wait-set*. Quando si chiama wait su un oggetto il thread viene sospeso e inserito nel *wait-set* di quell'oggetto. Quando il *wait-set* riceve una notifica con `notify()`, uno degli oggetti in attesa viene risvegliato e messo in coda pronti. Se invece la notifica è con `notifyAll()`, vengono risvegliati tutti.