# 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*.

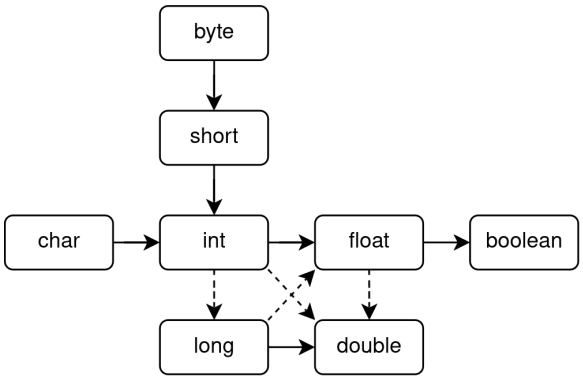
Vediamo poi che la parola chiave `const` non è implementata in Java (anche se è comunque riservata), mentre sono disponibili le parole chiave `static` e `final`. In particolare questa assegna a variabili valori finali, cioè che non possono più essere alterati.

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`:

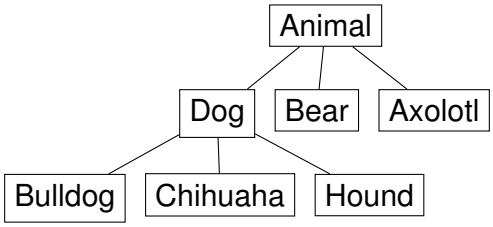
```
class Animal {
  // implementazione di Animal
}

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).

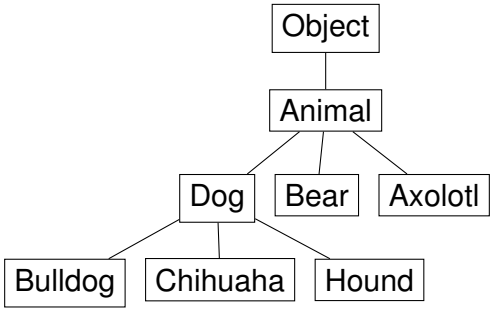
La parole chiave `super` può essere usata anche per chiamare il costruttore della classe padre (`super()`). In questo modo possiamo inizializzare la sezione di membri della classe padre quando si costruisce la classe figlia.

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:

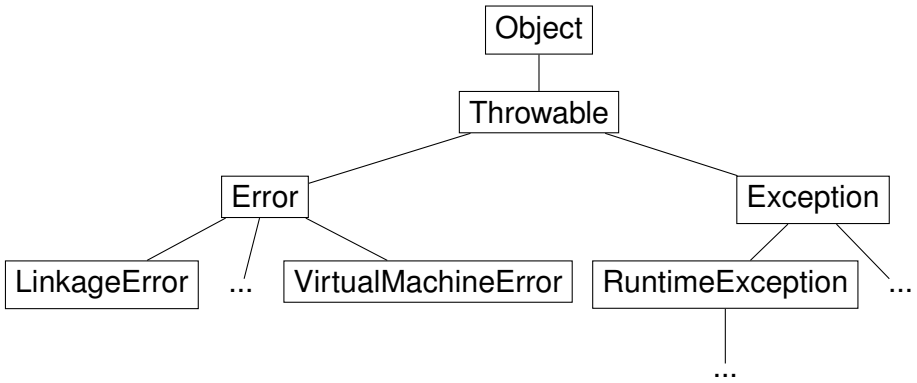
```
enum Colori {
  ROSSO,
  VERDE,
  GIALLO
}
```

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).

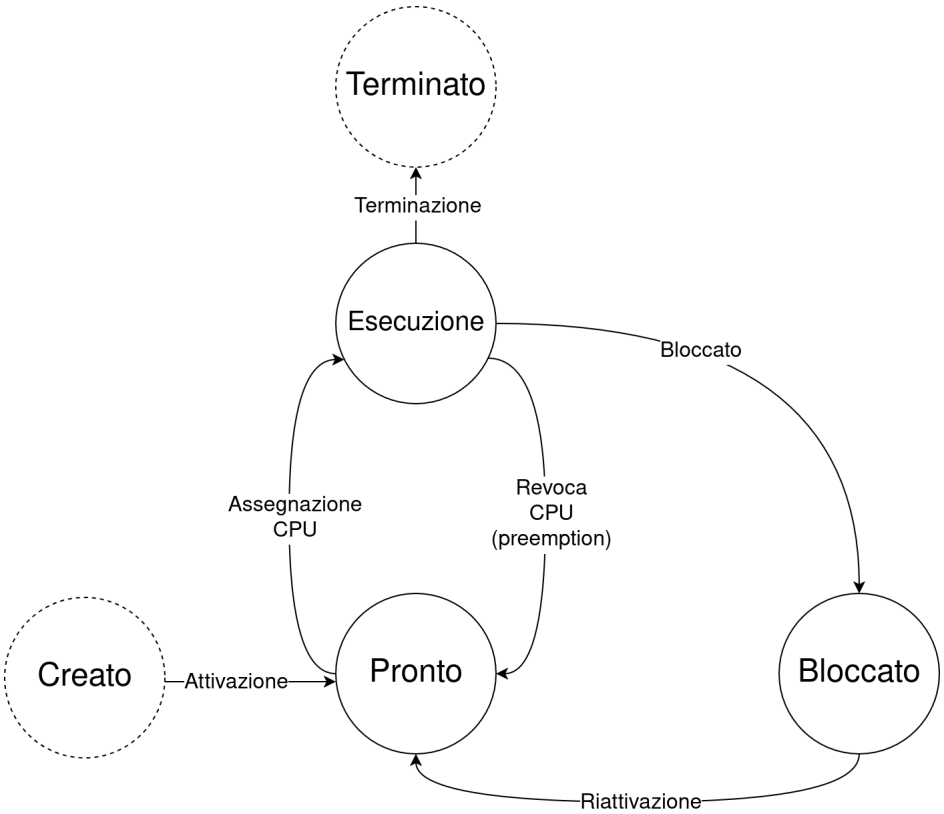
Come per la classe Thread, l'interfaccia Runnable è provvista di un metodo astratto `run()` da ridefinire per specificarne il comportamento.

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.

## 7 Lezione del 21-10-25

Continuiamo a parlare della sincronizzazione, approfondendo il funzionamento dei metodi relativi.

#### 7.0.1 Metodo `wait()`

Il metodo `wait()` è definito come `public final void wait() throws InterruptedException`. Il thread che invoca `wait()`:

- Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Riacquisisce automaticamente il lock quando esece dalla wait.

Esiste una versione con timeout di `wait()`. Questa è definita come `public final void wait(long timeout) throws InterruptedException`. In questo caso il thread che invoca `wait()`:

- Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Aspetta i millisecondi di timeout: se li supera esce forzatamente riacquisendo i lock.

La `wait(timeout)` con `timeout` $= 0$ corrisponde alla `wait()` (timeout infinito).

### 7.0.2 Metodo `notify()`

Il metodo `notify()` ha firma `public final void notify()`. Quando viene invocato su un oggetto il primo fra i thread che erano in attesa su tale oggetto (cioè erano nella sua *wait list*, dove si entra invocando `wait()`) viene svegliato.

Il `notifyAll()` è una variante di `notify()`, con firma `public final void notifyAll()`, che sveglia tutti i thread in attesa nella *wait list*.

Notiamo che `wait()`, `notify()` e `notifyAll()` non devono essere per forza chiamati sulla classe madre, ma possono anche essere chiamati su altri oggetti, con la prerogativa che si possieda un lock su tali oggetti.

## 7.1 Produttori e consumatori

Vediamo come applicare questo schema ha uno degli esempi più noti della programmazione concorrente: quello del paradigma dei **produttori** e **consumatori**.

Ipotizziamo una situazione dove esistono $n$ thread, $P_0$, $P_1$, ..., $P_n$ detti *produttori*, ed $m$ thread, $C_0$, $C_1$, ..., $C_m$ detti *consumatori*. Fra produttori e consumatori c'è un certo *buffer*. Il buffer è *condiviso* tra tutti i produttori e i consumatori, ed è in grado di contenere un numero limitato di valori.

Ogni produttore ciclicamente:

- Produce un nuovo valore;
- Lo immette nel buffer.

Chiaramente, se il buffer è pieno, il produttore deve bloccarsi.

Ogni consumatore ciclicamente:

- Preleva un valore dal buffer;
- Lo consuma.

Come sopra, se il buffer è vuoto, il consumatore deve bloccarsi.

Vediamo quindi l'implementazione di questo pattern:

```
1  class Buffer {
2    final int buf[];
3    final int size;
4    int beg;
5    int end;
6
7    Buffer(int size) {
8      this.size = size;
9      buf = new int[size];
10   }
11
12   boolean isFull() {
13     return (end + 1) % size == beg;
14   }
15
16   boolean isEmpty() {
17     return end == beg;
18   }
19
20   synchronized void insert(int e) {
21     while(isFull()) {
```

```
22        System.out.println("insert() bloccato da buffer pieno");
23        try {
24          wait();
25        } catch(InterruptedException ex) {}
26      }
27
28      System.out.println("insert() di " + e);
29      buf[end] = e;
30      end = (end + 1) % size;
31      notifyAll();
32    }
33
34    synchronized int extract() {
35      while(isEmpty()) {
36        System.out.println("extract() bloccato da buffer vuoto");
37        try {
38          wait();
39        } catch(InterruptedException ex) {}
40      }
41
42      int e = buf[beg];
43      beg = (beg + 1) % size;
44      System.out.println("extract() di " + e);
45      notifyAll();
46
47      return e;
48    }
49  }
50
51  class Producer extends Thread {
52    static int c = 0;
53
54    Buffer b;
55    String name;
56
57    Producer(String name, Buffer b) {
58      this.name = name;
59      this.b = b;
60    }
61
62    public void run() {
63      while(true) {
64        // aspetta un tempo casuale
65        try {
66          sleep((long) (Math.random() * 1000));
67        } catch(InterruptedException ex) {}
68
69        // inserisci
70        System.out.println(name + " ha prodotto " + c);
71        synchronized(Producer.class) {
72          b.insert(c++);
73        }
74      }
75    }
76  }
77
78  class Consumer extends Thread {
79    Buffer b;
80    String name;
81
```

```
82    Consumer(String name, Buffer b) {
83      this.name = name;
84      this.b = b;
85    }
86
87    public void run() {
88      while(true) {
89        // aspetta un tempo casuale
90        try {
91          sleep((long) (Math.random() * 1000));
92        } catch(InterruptedException ex) {}
93
94        // estrai
95        int c = b.extract();
96        System.out.println(name + " ha consumato " + c);
97      }
98    }
99  }
100
101 class Main {
102   public static void main(String[] args) {
103     // crea buffer
104     Buffer buf = new Buffer(10);
105
106     // popola produttori
107     for(int i = 0; i < 25; i++) {
108       new Producer("Produttore " + i, buf).start();
109     }
110
111     // popola consumatori
112     for(int i = 0; i < 15; i++) {
113       new Consumer("Consumatore " + i, buf).start();
114     }
115
116     // simula per un po'
117     try {
118       Thread.sleep(5000);
119     } catch(InterruptedException ex) {}
120
121     // termina
122     System.exit(0);
123   }
124 }
```

### 7.2 Differenza fra `notify()` e `notifyAll()`

Approfondiamo la differenza fra i metodi per il risveglio di thread, cioè `notify()` e `notifyAll()`.

Su un oggetto possono essere in attesa più thread (alternativamente, in una *wait list* possono esserci più di un thread).

Se questi thread sono in attesa di condizioni diverse, è opportuno usare la `notifyAll()`. In tal situazione, se si facesse diversamente (usando una semplice `notify()`) si potrebbe infatti avere la sveglia di un solo thread che però non vede le proprie condizioni soddisfatte: nessuno viene effettivamente svegliato, si può incorrere in deadlock.

Se tutti i thread bloccati attendono la stessa condizione, o se solo uno (senza specificare quale) dei thread bloccati può trarre beneficio dalla situazione creatasi, conviene invece usare `notify()`.

Nell'esempio sopra, sul buffer si usa `notifyAll()` anziché `notify()` in quanto produttori e consumatori possono entrambi bloccarsi sui thread, ma aspettando condizioni diverse.

# 8 Lezione del 23-10-25

## 8.1 Gestore di risorse equivalenti

Vediamo un altro pattern tipico della programmazione concorrente la **gestione di risorse equivalenti**.

Assumiamo di avere un insieme finito di **risorse**, gestite da un certo **gestore**. Il gestore ha il compito di offrire due metodi:

- `int richiesta()`: restiuisce l'indice della risorsa che viene attribuita al chiamante. Può venire bloccata se non ci sono risorse rimanenti;
- `void rilascia(int k)`: rilascia la risorsa $k$-esima (che dovrebbe essere quella assegnata al chiamante).

Potremmo implementare tale sistema come segue:

```
class Handler {
  boolean[] assigned;
  int numAssigned;

  public Handler(int n) {
    assigned = new boolean[n];
  }

  public synchronized int request() throws InterruptedException {
    while(numAssigned == assigned.length) {
      wait();
    }

    int i = 0;
    while(i < assigned.length && assigned[i]) i++;
    assigned[i] = true;

    numAssigned++;
    return i;
  }

  public synchronized void release(int k) throws InterruptedException {
    assigned[k] = false;
    numAssigned--;

    // le attese sono tutte equivalenti
    notify();
  }
}

class User extends Thread {
  String name;
  Handler h;
  int numRequests;

  public User(String name, Handler h, int numRequests) {
    this.name = name;
```

```
38      this.h = h;
39      this.numRequests = numRequests;
40    }
41
42    public void run() {
43      try {
44        for(int i = 0; i < numRequests; i++) {
45          // ottieni
46          sleep((long) (Math.random() * 1000));
47          int k = h.request();
48          System.out.println(name + " ha ottenuto " + k);
49
50          // rilascia
51          sleep((long) (Math.random() * 1000));
52          h.release(k);
53          System.out.println(name + " ha rilasciato " + k);
54        }
55      } catch(InterruptedException e) {
56        System.err.println(name + " e' stato interrotto");
57      }
58    }
59  }
60
61  class Main {
62    public static void main(String[] main) {
63      Handler h = new Handler(10);
64
65      for(int i = 0; i < 10; i++) {
66        (new User("Utilizzatore " + i, h, 100)).start();
67      }
68    }
69  }
```

## 8.2 Metodo `join()`

Il metodo `join()` viene offerto dalla classe *Thread*, e serve ad aspettare che un altro thread finisca.

Viene chiamato su un istanza di oggetto di tipo thread (non corrispondente al thread corrente), e ha il seguente effetto:

- Il thread corrente si blocca finché il thread riferito non termina;
- Se il thread riferito ha già terminato, non ha effetto.

Esistono 3 versioni del `join()`:

- `public final void join() throws InterruptedException`: come quello appena discusso;
- `public final void join(long millis) throws InterruptedException`: come sopra, ma prevede un timeout di `millis` millisecondi;
- `public final void join(long millis, int nanos) throws InterruptedException`: di nuovo come sopra, ma prevede un timeout di `millis` millisecondi e `nanos` nanosecondi;

# 9 Lezione del 30-10-25

## 9.1 Interrompere un thread

Nelle prime versioni di Java esisteva un metodo sui thread, lo `stop()`, che permetteva al chiamante di arrestare immediatamente l'esecuzione di un thread. Questo meccanismo è stato deprecato quasi subito, in quanto pone seri rischi (ad esempio ci lascia facilmente lasciare strutture dati in stato inconsistente).

Quello che si consiglia di fare è quindi semplicemente prevedere variabili `boolean running` nei thread, aggiornate da fuori ai thread, che la `run()` dei thread controlla periodicamente per arrestarsi in maniera autonoma e pulita.

### 9.1.1 Interruzioni thread

Si possono mandare **interrupt** sui thread, usando il meotodo `interrupt()`. Quando si invia un interrupt, può accadere una di 2 cose:

1. Se il thread è bloccato con una `sleep()`, una `wait()` o una `join()`, il thread esce e entra nei rispettivi blocchi `catch` con l'*InterruptedException*. In questo caso l'*interrupt status* non viene controllato;
2. Se il thread è in normale esecuzione, l'*interrupt status* viene controllato.

L'**interrupt status** è una condizione che può essere valutata dal thread con 2 metodi:

- `public static boolean interrupted()`: restituisce l'interrupt status e lo resetta;
- `public boolean isInterrupted()`: restituisce l'interrupt status senza resettare.

## 9.2 Classi wrapper

Abbiamo introdotto in 1.3 le classi **wrapper** per i tipi primitivi. Queste sono, in particolare:

| Wrapper | Tipi |
|---|---|
| Boolean | `boolean` |
| Byte | `byte` |
| Short | `short` |
| Int | `int` |
| Long | `long` |
| Float | `float` |
| Double | `double` |
| Character | `char` |

Le classi wrapper possono essere create via `new`, o molto più spesso attraverso il metodo `valueOf()` (che accetta anche stringhe). Sono **immutabili**, e perciò non possono cambiare (operazioni danno nuovi oggetti).

Solitamente rappresentano anche un modo per raggruppare costanti semanticamente coerenti col tipo rappresentato (per gli interi, *intero minimo*, *intero massimo*, ecc...).

### 9.2.1 Autoboxing / unboxing

I meccanismi di **autoboxing** e *(auto)***unboxing** permettono di convertire automaticamente da tipi primitivi a classi wrapper, senza aver bisogno di usare i metodi `valueOf()`.

In particolare:

- Il *boxing* avviene quando una variabile o un argomento formale di tipo classe wrapper viene inizializzato con una corrispondente variabile di tipo primitivo: `Integer i = 10;`
- L'*unboxing* avviene, viceversa, quando una variabile o un argomento formale di tipo primitivo viene inizializzato con una corrispondente variabile di tipo classe wrapper: `int i = Integer.valueOf(10).`

## 9.3 JMM

Il **JMM** (*Java Memory Model*) è il modello secondo cui Java mantiene le variabili (i campi) delle classi. Sostanzialmente, è l'insieme di regole che stabiliscono l'ordinamento degli accessi alla memoria e quando le modifiche sono visibili in modo garantito. Ci è di interesse quando si parla di variabili condivise fra classi.

In Java, infatti, ogni valore condiviso fra thread deve essere acceduto in mutua esclusione (con `synchronized`) per evitare interferenze.

L'acquisizione di un lock ha un costo, e in alcuni casi potrebbe non essere necesario: il linguaggio garantisce che le operazioni di lettura /scrittura su variabili (eccetto double e long) sono atomiche. Quindi, se c'è un thread che modifica una variabile e altri che la leggono non ci sono problemi di interferenza. Questo però non assicura che un thread legga il valore più recente di una variabile (valore scritto da un altro thread), in assenza di meccanismi di sincronizzazione. Diversi fattori influiscono su quando una variabile modificata da un thread appare come tale ad altri thread.

Il linguaggio prevede che certe "azioni" siano caratterizzate da una relazione "**happens-before**" ("*avviene prima*"):

- Il rilascio di un lock su un oggetto "avviene prima" di ogni successiva acquisizione del lock sullo stesso oggetto;
- La scrittura su una variabile volatile "avviene prima" di ogni successiva lettura della stessa variabile;
- La chiamata a `start()` "avviene prima" delle azioni nel thread che è stato fatto partire;
- Le operazioni in un thread "avvengono prima" della join() su tale thread.

# 10 Lezione del 04-11-25

## 10.1 Java moderno

Vediamo quindi alcune funzionalità introdotte e in uso nelle versioni più recenti di Java.

Non notiamo alcuni dettagli più piccoli, come gli switch che supportano stringhe e le condizioni multiple nei blocchi `try` (usando l'operatore |).

### 10.1.1 Annotazioni

Le **annotazioni** sono uno strumento introdotto in Java 5 che si affianca ai commenti. La differenza fra commenti e annotazioni è che le annotazioni sono strutturate.

Esistono alcune annotazioni predefinite:

- *@Deprecated*, significa che un metodo è deprecato e quindi bisognerebbe cercare di non usarlo;
- *@Override*, significa che un metodo ridefinisce uno della superclasse;
- *@SuppressWarnings*, sopprime avvisi dal compilatore.

Le annotazioni possono essere definite dal programmatore, sfruttando una sintassi simile alla dichiarazione di interfacca (ma usando la chiocciola) ad esempio:

```
import java.lang.annotation.*;
[...]

// visibile a runtime (attraverso reflection)
@Retention(RetentionPolicy.RUNTIME)
// applicabile a metodi
@Target(ElementType.METHOD)
public @Interface Info {
  String info1();
  String info2();
}
```

Vediamo come si prevedono due annotazioni preliminari alla definizione di annotazione:

- *@Retention*, indica dove l'annotazione è visibile fra *SOURCE* (solo nel sorgente, scartata dal compilatore), *CLASS* (presente nel bytecode ma non visibile a runtime), *RUNTIME* (accessibile tramite reflection). Nell'esempio è `RUNTIME`;
- *@Target*, indica l'obiettivo dell'annotazione (metodi, classi, ecc...). Nell'esempio è *METHOD* (metodo).

Si può quindi usare l'annotazione dichiarata come segue:

```
class MiaClasse {
  @Info(info1 = "qualcosa", info2 = "qualcos'altro")
  public void foo() {
    // ...
  }
}
```

A questo punto si può, attraverso reflection, accedere all'annotazione di `foo()`:

```
import java.lang.reflect.*;
[...]

Method method = MiaClasse.class.getMethod("foo");
if (method.isAnnotationPresent(Info.class)) {
    Info annotation = method.getAnnotation(Info.class);

    // stampa i campi dell'annotazione
    System.out.println(annotation.info1());
    System.out.println(annotation.info2());
}
```

### 10.1.2 Reflection

La **reflection** (*riflessione*) è il meccanismo che permette ai programmi Java di ispezionare altri programmi Java (incluso il programma stesso).

Questo avviene attraverso classi disponibili in `java.lang.reflect`:

- **Class**: è una classe le cui istanze rappresentano altre classi;
- **Method**: è una classe le cui istanze rappresentano metodi di altre classi;
- **Field**: è una classe le cui istanze rappresentano campi di altre classi.

Esistono poi altre classi che rappresentano costruttori, ecc...

Abbiamo già visto un esempio di reflection nella scorsa sezione riguardante le annotazioni. Vediamone un'altro:

```
import java.lang.reflect.*;
[...]

Object cane = new Cane();
Class<?> clazz = cane.getClasse();

// da qui possiamo accedere alla struttura della classe Cane, e.g.:
Field[] fields = clazz.getDeclaredFields();
List<String> actualFields = getFieldNames(fields);
System.out.println(actualFields.get(0)); // magari "nome"
```

### 10.1.3 Import statici

Java permette, sempre dalla versione 5, di fare i cosiddetti **import statici**, cioè importare tutti i metodi e campi statici di una classe. Questo si fa semplicemente aggiungendo la parola chiave `static` a le direttive `import`. Una volta fatto lo static import da una classe, si possono usare gli identificatori statici di quella classe senza prefiggere il nome della classe.

### 10.1.4 Classi innestate

Approfondiamo il discorso delle classi innestate.

Le classi statiche possono essere innestate all'interno di altre classi. In questo caso:

- Possono accedere a tutti i campi statici della classe esterna, anche privati;
- Possono accedere a tutti i campi della classe esterna, anche privati, se ne hanno un riferimento.

Le interfacce e gli enum annidati a classi sono implicitamente static.

Le classi innestate non statiche sono dette *classi interne*.

- Ogni istanza di classe interna è associata ad una classe esterna, ed ha accesso ai riferimenti `this` e `Esterna.this` (cioè accesso alla classe esterna qualificando il `this` con il nome della classe esterna);
- Le classi interne non possono avere membri statici;
- Possono invece accedere ai campi privati della classe esterna: se non ci sono ambiguità, infatti, il riferimento this sia a sé stesse che alla classe esterna può essere omesso.

### 10.1.5 Metodi default

I **metodi di default** servono a coprire un buco lasciato dalle interfacce. Quando si definisce un metodo all'interno di un'interfaccia, infatti, server che ogni classe che implementa tale interfaccia implementi la sua versione di tale metodo. Un metodo dichiarato come `default` all'interno di un'interfaccia, invece, permette di specificare un'implementazione predefinita, che viene quindi usata da tutte le interfacce (può eventualmente essere ridefinita).

Il problema è chiaramente la sovrapposizione che si ha data dalla definizione e implementazione in più interfacce (o anche solo in una classe base e quindi in interfacce) di una classe. In questo caso si deve usare il nome completamente qualificato del metodo (cioè riferirsi esattamente alla classe o all'interfaccia la cui versione del metodo vogliamo).

Notiamo a questo punto che si possono definire anche metodi statici all'interno di interfacce (senza doverli specificare come `default`). Questi sono metodi implementati direttamente nelle interfacce (tipicamente di utilità), legati all'interfaccia e non disponibili a classi che non implementano tale interfaccia.

### 10.1.6 Argomenti variabili

Java permette di avere metodi con numero di argomenti variabile (*varargs*) usando la solita sintassi coi tre puntini. Gli argomenti risultano quindi accessibili come vettori. Ad esempio, si può avere:

```
void metodo(String...a) {
  for(String x : a) {
    // stampa tutti gli argomenti forniti
    System.out.println(x);
  }
}

[...]

// entrambi validi
metodo("Ciao");
metodo("Ciao", "mondo");
// ecc...
```

### 10.1.7 Try-with-resources

Una soluzione introdotta in Java 7 è quella delle **try-with-resources**, usata quando bisogna accedere a risorse, gestire delle eccezioni e quindi liberare tali risorse in presenza di eccezioni.

La sintassi storica per questo tipo di operazione era la seguente:

```
void read(String n) throws IOException {
  FileReader f = new FileReader(n);
  BufferedReader b = new BufferedReader(f);
  try {
    System.out.println(b.readLine());
  } finally {
    b.close();
    f.close();
  }
}
```

Notando che andiamo subito al blocco `finally`, in quanto per qualsiasi eccezione vogliamo comunque fare le stesse cose: chiudere `b` ed `f` (nell'ordine inverso a quello in cui vengono creati). Questo approccio può però portare a leak di risorse: poniamo ad esempio che la `b.close()` generi a sua volta un'eccezione: in questo caso la `f.close()` non viene mai eseguita ed il file non viene liberata.

Da Java 7 si può risolvere questo problema come segue

```
void read(String n) throws IOException {
  try(FileReader f = new FileReader(n); BufferedReader b = new
    BufferedReader(f)) {
    System.out.println(b.readLine());
  }
  // qui f e b sono stati chiusi!
}
```

In questo caso il meccanismo delle try-with-resources ci assicura che le risorse verranno chiuse automaticamente, e in ogni caso, nell'ordine inverso a quello in cui vengono acquisite (cioè dichiarate nella condizione del `tr`).

## 10.2 Tipi generici

I **tipi generici** sono simili ai template del C++. Usare i tipi raw (come ad esempio *Object*) può portare a problemi non rilevati in fase di compilazione come ad esempio conversioni invalide, ecc... Ad esempio, potremmo avere:

```
public class Casella {
  private Object o;

  public Casella(Object o) {
    this.o = o;
  }

  public Object prendi() {
    return o;
  }

  public void copia(Casella c) {
    this.o = c.o;
  }
}

[...]

Casella c1 = new Casella("ABC");
Casella c2 = new Casella(Integer.valueOf(123));
c1.copia(c2);
String s = (String) c1.prendi(); // errore! int non puo' essere convertito
    in stringa
```

La soluzione è rappresentata dai tipi generici. Un tipo generico è una classe o un'interfaccia parametrizzata in termini di tipi. Ad esempio, si può avere:

```
public class Casella<T> {
  private T t;

  public Casella(T t) {
    this.t = t;
  }

  public Object prendi() {
```

```
    return t;
  }

  public void copia(Casella<T> c) {
    this.t = c.t;
  }
}
```

In questo caso, attraverso la notazione <>, si riesce a discriminare fra classi *Casella* sulla base del tipo di dati che trasportano. Si avrà quindi rilevamento di errori come quello riportato prima, ma a tempo di compilazione:

```
Casella c1 = new Casella("ABC");
Casella c2 = new Casella(Integer.valueOf(123));
c1.copia(c2); // errore! Casella<Integer> non puo' essere convertito in
    Casella<String>
```

Per i tipi generici si può usare la wildcard <?>, già vista in 10.1.2 quando stavamo parlando della riflessione. Un riferimento con tipi generici wildcard non può essere istanziato, ma può essere usato per riferire a classi con tipi generici reali.

Ad esempio:

```
w = Casella<?>;

cs = new Casella<>("ABC");
ci = new Casella<>(Integer.valueOf(123));

w = ci;
w = cs; // permesso

String s = (String) c1.prendi();
```

Questo chiaramente ci riporta alla situazione con cui abbiamo aperto la sezione: questo non ci turba, in quanto con gli wildcard dobbiamo esplicitamente specificare che vogliamo riferirci a classi con tipi generici diversi fra di loro (quindi ci portiamo in una situazione dove siamo soggetti ad errori).

Notiamo poi che le wildcard possono essere anche più specifiche: in particolare, si possono usare wildcard con classi base per riferirsi a tipi generici classi specializzate.

# 11 Lezione del 06-11-25

Continuiamo con la discussione di costrutti avanzati del linguaggio Java.

### 11.0.1 Classi anononime

Le classi **anonime** sono un costrutto del linguaggio che permette di specializzare classi parent direttamente quando si instanzia la specializzazione. La sintassi è la seguente:

```
// class Libro ...
Libro libroJava = new Libro("Java") {
    @Override
    public String descrizione() {
        return "Libro che fa venire gran mal di testa.";
    }
}
```

In questo caso la classe figlia che specializza *Libro* viene dichiarata ed instanziata contestualmente. La classe è effettivamente anonima in quanto non si può riferire da altre parti nel codice.

### 11.0.2 Espressioni lambda

Le **espressioni lambda** o *funzioni lambda* sono supportate in Java attraverso la sintassi a freccia tipica del JavaScript.

In particolare, si può dichiarare una funzione lambda catturando il contesto e quindi definendo la funzione con la freccia ->:

```
List<String> l = new ArrayList<String>();
// ...
l.forEach((s) -> System.out.println(s));
```

Nell'esempio, il metodo `forEach()` accetta come argomento una funzione lambda, che accetta come argomento una stringa (catturata in `s`). In seguito, esegue tale funzione su ogni elemento della lista, ogni volta collassando `s` all'elemento corrente della lista.

Espressioni lambda e classi anonime sono fra di loro molto simili. Effettivamente, il comportamento dell'ultimo esempio si potrebbe avere come:

```
List<String> l = new ArrayList<String>();
// ...
l.forEach(new Consumer<String>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
});
```

In questo caso il metodo `forEach()` accetta anzichè un funzionale lambda, un oggetto di tipo *Consumer*. Questo si comporta come una lambda, accentando un argomento attraverso il metodo `accept()` che viene ridefinito contestualmente all'instanziazione attraverso una classe anonima.

## 11.1 Stream

In Java 8 viene implementata l'interfaccia *Stream*. Questa server ad astrarre oggetti di tipo **stream**, cioè sequenze di elementi che supportano operazioni.

Le operazioni sugli stream in Java possono essere *sequenziali* e *parallele*: di default si assumono seriali, se richiesto possono diventare parallele.

L'approccio agli stream è *funzionale*: sono definite funzioni che prendono in argomento stream, e restituiscono stream o altri risultati. Questo significa che lo stato di ogni stream non viene mutato, ma se ne creano di nuovi.

Le operazioni sugli stream possono essere di 2 tipi:

- Operazioni **intermedie**: danno come risultato altri stream. Queste possono avere stato (come ad esempio le operazioni di sorting) o non avere stato;
- Operazioni **terminali**: danno un risultato finale, e non altri stream su cui elaborare.

Spesso si usano funzioni lambda per modificare il comportamento delle operazioni. Un'esempio di come possono essere usati gli stream è il seguente:

```
Stream<Elem> s = Stream.of(/* ... */);
long res = s
  .skip(10)                          // salta 10 elementi
  .filter(e -> e.getValore() > 0.5) // filtra quelli > 0.5
  .map(e -> e.getValore() + 1)      // mappali agli elementi + 1
  .distinct()                        // prendi elementi distinti
  .count()                           // conta gli elementi
```

Gli stream possono essere creati a partire da array, liste o tipi insieme simili.

## 11.2 GC

Veniamo quindi alla trattazione del **GC** (*Garbage Collector*) in Java. Questo è un componente che si occupa di liberare, al posto nostro, la memoria che non utilizziamo più.

Il garbage collector può essere chiamato direttamente dal programma accedendo ad un metodo della classe *System*:

```
// chiama il GC
System.gc();
```

Questo invoca effettivamente il GC, e solitamente non è necessario (in quanto il GC automaticamente viene messo in esecuzione quando necessario).

### 11.2.1 GC generazionali

La maggior parte delle implementazioni di JVM odierne sfrutta i cosiddetti GC **generazionali**.

Gli oggetti si dividono effettivamente in 2 categorie:

- Oggetti che vivono poco;
- Oggetti che vivono molto.

I GC generazionali si basano sull'idea di dividere in *generazioni* gli oggetti creati sulla base del tempo da cui sono stati instanziati, o equivalentemente il tempo per cui sono *sopravvissuti* all'interno del nostro programma.

Si adotta quindi un sistema simile alle code multiple dove:

1. Gli oggetti appena creati vengono messi nella **YG** (*Young Generation*);
2. Dopo un certo tempo, gli oggetti sopravvissuti della YG vengono spostati nella **OG** (*Old Generation*);
3. Infine, si prevede una **PG** (*Permanent Generation*) per gli oggetti permanenti (cioò che non verrranno mai deallocati).

A questo punto la garbage collection può essere effettuata come segue:

- **Minor** garbage collection, più frequente e solo sulla YG;
- **Major** garbage collection, meno frequente (quando lo spazio dedicato all'OG finisce) e solo sulla OG.

Lo spazio dove vengono allocati gli oggetti della YG viene detto **eden**. Gli oggetti che sopravvivono alla minor garbage collection vengono quindi spostati nello spazio **survivor**. Infine, gli oggetti della OG vengono allocati nello spazio **old** (o *tenured*).

### 11.2.2 Esecuzione "stop the world"

Le operazioni del GC sono solitamente di tipo **"stop the world"**, cioè interrompono l'intera esecuzione del programma.

Anche per questo motivo si decide di divedere fra minor garbage collection e major garbage collection: avere garbage collection di oggetti grandi troppo spesso porterebbe a rallentamenti considerevoli del programma in esecuzione.

In ogni caso, per visualizzare statistiche sull'operato del GC (ad esempio dimensione della memoria libera nell'eden o le altre pool di memoria) si possono usare programmi di profiling. Uno dei più celebri è *VisualVM*.

## 11.3 Design pattern

I **design pattern** sono, appunto, pattern che emergono spesso in fase di progettazione, sopratutto nell'ambito dell'OOP.

Si dividono in:

- **Creazionali**: legati al modo con cui vengono istanziati gli oggetti;
- **Strutturali**: focalizzati sulla composizione e sulle relazioni tra oggetti e classi;
- **Comportamentali**: riguardano il modo con cui gli oggetti interagiscono fra di loro e si scambiano informazioni.

### 11.3.1 Pattern creazionali

I pattern **creazionali** riguardano il modo in cui vengono creati gli oggetti. Questi sono:

- **Singleton**: un oggetto che deve esistere in unica istanza in tutto il sistema, e fornire un modo per accedere a tale istanza. L'idea centrale è impedire la creazione di più istanze di una stessa classe in contesti dove la molteplicità non ha senso, come un gestore delle configurazioni, un logger centralizzato o un registro delle connessioni attive.

  Vediamo ad esempio una classe logger:

```
public class Logger {
  private Logger() {
    // qui si inizializza (1 volta) il logger
  }

  private static class Holder {
    private static final Logger instance = new Logger();
  }

  public static Logger getInstance() {
    return Holder.instance;
  }

  public void log() {
    // ...
  }
}
```

  Notiamo come il modo in cui implementiamo l'unicità del singleton è differente da quella che avevamo visto in 2.5. Lì, infatti, avevamo usato un riferimento esplicito e statico ad un istanza di classe che gestivamo. Qui scegliamo invece di usare una classe statica innestata, che mantiene un'istanza finale di classe *Logger*.

  Questa soluzione ci permette di ottenre lo stesso comportamento, ma in maniera *thread-safe*: possiamo infatti usare il meccanismo di caricamento della JVM stessa per realizzare l'instanziazione condizionale della classe.

- **Factory method**: un interfaccia per creare oggetti, che lascia che le sottoclassi decidano quale oggetto istanziare. Sostanzialmente è un modo per lasciare che le sottoclassi (o le classi che implementano interfacce) si occupino dell'instanziazione della classe padre. Risulta utile quando la classe che deve creare un oggetto non conosce in anticipo il tipo esatto da istanziare oppure desidera lasciare alle sottoclassi

il compito di decidere quale tipo restituire. Inoltre, permette il disaccoppiamento tra il codice che richiede l'oggetto e il codice che lo costruisce.

Vediamo ad esempio un interfaccia *Notify*, per notifiche di vario tipo, che viene specializzata in *SmsNotification* e *EmailNotification*. Prevediamo di usare un metodo fabbrica per gestire l'istanziazione di questo tipo di notifiche:

```
interface Notify {
  void send(String msg);
}

// prodotto concreto
class SmsNotification implements Notify {
  @Override
  public void send(String msg) {
    System.out.println("SMS sent: " + msg);
  }
}
// prodotto concreto
class EmailNotification implements Notify {
  @Override
  public void send(String msg) {
    System.out.println("Email sent: " + msg);
  }
}

// interfaccia di creazione
interface NotificationCreator {
  Notify createNotification(); // factory method
}

// concrete creator
class EmailCreator implements NotificationCreator {
  @Override
  public Notify createNotification() { // implementazione factory
    method
    return new EmailNotification();
  }
}
// concrete creator
class SmsCreator implements NotificationCreator {
  @Override
  public Notify createNotification() { // implementazione factory
    method
    return new SmsNotification();
  }
}

public class Example {
  public static void main(String[] args) {
    NotificationCreator creator = new EmailCreator();
    Notify notif = creator.createNotification();
    notif.send("Login to your account");
    // user switches to sms notifications
    creator = new SmsCreator();
    notif = creator.createNotification();
    notif.send("Password changed");
  }
}
```

- **Abstract factory**: qui l'idea è di fornire un interfaccia per la realizzazione di famiglie di oggetti correlati o dipendenti fra di loro, senza dover specificare le loro classi concrete.

  Risulta un paradigma utile in contesti in cui le classi concrete da utilizzare cambiano in blocco, e si vuole quindi delegare la creazione non di un singolo oggetto, ma di una loro famiglia, a un singolo *factory method*.

  Vediamo quindi un esempio di due oggetti, che vogliamo poter renderizzare su PDF e su HTML:

```
// istanze astratte

interface Paragraph {
  void show();
}

interface Table {
  void show();
}

// istanze concrete PDF

class ParagraphPDF implements Paragraph {
  void show() {
    // ...
  }
}

class TablePDF implements Table {
  void show() {
    // ...
  }
}

// istanze concrete HTML

class ParagraphHTML implements Paragraph {
  void show() {
    // ...
  }
}

class TableHTML implements Table {
  void show() {
    // ...
  }
}
```

  Per creare un documento, vorremo quindi usare più factory relative a documenti PDF e HTML, che ineritano dalla stessa *abstract factory*:

```
// costruttore astratto
interface DocumentFactory {
  Paragraph createParagraph();
  Table createTable();
}

// costruttore concreto PDF
class DocumentFactoryPDF implements DocumentFactory {
  Paragraph createParagraph() {
```

```
    return new ParagraphPDF();
  }
  Table createTable() {
    return new TablePDF();
  }
}

// costruttore concreto HTML
class DocumentFactoryHTML implements DocumentFactory {
  Paragraph createParagraph() {
    return new ParagraphHTML();
  }
  Table createTable() {
    return new TableHTML();
  }
}
```

A questo punto creare un nuovo documento di tipo PDF o HTML sarà molto semplice, in quanto basterà dire:

```
class Document {
  private Paragraph p;
  private Table t;

  public Document(DocumentFactory factory) {
    p = factory.createParagraph();
    t = factory.createTable();
  }

  public void show() {
    p.show();
    t.show();
  }
}

// creiamo un documento PDF
DocumentFactory factory = new DocumentFactoryPDF();
Document doc = new Document(factory);
```

- **Builder**: consiste nel separare la costruzione di un'oggetto complesso dalla sua rappresentazione. Risulta utile quando è necessario fornire un numero molto grande di parametri in fase di costruzione.

  Vediamo ad esempio una classe computer piuttosto complessa:

```
class Computer {
  private final String cpu;
  private final int ram;
  private final boolean hasBluetooth;
  private final boolean hasWifi;

  // costruttore privato
  private Computer(ComputerBuilder builder) {
    this.cpu = builder.cpu;
    this.ram = builder.ram;
    this.hasBluetooth = builder.hasBluetooth;
    this.hasWifi = builder.hasWifi;
  }

  public void mostra() {
    System.out.println("CPU: " + cpu);
```

```
    System.out.println("RAM: " + ram);
    System.out.println("Bluetooth: " + (hasBluetooth ? "Y" : "N"));
    System.out.println("Wi-Fi: " + (hasWifi ? "Y" : "N"));
  }
}
```

ed implementiamo una classe builder che ne semplifichi l'instanziazione:

```
// classe builder annidata statica
public static class ComputerBuilder {
  private final String cpu;
  private final int ram;
  private boolean hasBluetooth = false;
  private boolean hasWifi = false;

  public ComputerBuilder(String cpu, int ram) {
    this.cpu = cpu;
    this.ram = ram;
  }
  public ComputerBuilder conBluetooth() {
    this.hasBluetooth = true;
    return this;
  }
  public ComputerBuilder conWifi() {
    this.hasWifi = true;
    return this;
  }
  public Computer build() {
    return new Computer(this);
  }
}
```

A questo punto costruire un nuovo computer sarà molto semplice, in quanto ogni chiamata a un metodo di *ComputerBuilder* restituisce ancora un istanza di *ComputerBuilder*:

```
public class Esempio {
  public static void main(String[] args) {
    Computer pc =
    new Computer.ComputerBuilder("Intel i5", 16)
      .conBluetooth()
      .conWifi()
      .build();
      pc.mostra();
    }
}
```

- **Prototype**: qui vogliamo specificare i tipi di oggetti che vogliamo creare non come classi, ma sulla base di determinati oggetti predefiniti detti *prototipi*. In Java, questo è possibile attraverso l'interfaccia *Cloneable*, un interfaccia *marker* (lo è ad esempio anche *Serializable*), che segnala che un oggetto è clonabile attraverso il metodo `clone()` della classe *Object*. Notiamo che `clone()` fa la *shallow copy*.

  Vediamo un esempio anche di questo pattern:

```
public class Shape2D implements Cloneable {
  private int size;
  private String color;

  public Shape2D(int size, String color) {
```

```
    this.size = size;
    this.color = color;
  }

  public void draw() {
    System.out.println("Drawing a 2D shape with size "
      + size + " and color " + color);
  }

  @Override
  public Shape2D clone() throws CloneNotSupportedException {
    return (Shape2D) super.clone(); // shallow copy
  }

  public static void main(String[] args) {
    Shape2D original = new Shape2D(10, "red");
    try {
      Shape2D copy = original.clone();
      original.draw();
      copy.draw();
    } catch (CloneNotSupportedException e) {
      System.err.println("Error during cloning");
    }
  }
}
```

# 12 Lezione del 09-12-25

Continuiamo l'argomento dei design pattern.

### 12.0.1 Pattern strutturali

Dopo i pattern *creazionali*, iniziamo a parlare dei pattern **strutturali**. Questi riguardano la *composizione* di oggetti. Un principio spesso usato è il *composition over inheritance*, che consiglia di preferire la composizione di oggetti all'ereditarietà per ridurre l'accoppiamento tra le classi e migliorare la modularità dei sistemi.

Questi pattern, quindi, sono:

- **Adapter**: usato per convertire l'interfaccia di una classe in un'altra interfaccia, che chi usa la classe si aspetta. Risulta utile quando vogliamo riutilizzare classi esistenti all'interno di un nuovo sistema senza modificare né le classi originali né il codice che usa la classe.

  Vediamo ad esempio un lettore audio, per cui vogliamo adottare una certa classe *OldAudioReader* a supporto di una certa interfaccia *NewAudioReader*:

```
class OldAudioReader {
    public void play() {
      // ...
    }
}

interface NewAudioReader {
    void playMedia();
}

// adattatore
```

```
class OldAudioReaderAdapter implements NewAudioReader {
  private OldAudioReader reader;
  public OldAudioReaderAdapter(OldAudioReader reader) {
    this.reader = reader;
  }
  @Override
  public void playMediaMedia() {
    reader.play(); // adattiamo il metodo esistente
  }
}
```

- **Bridge**: l'obiettivo di questo pattern è disaccoppiare un'astrazione dalla sua implementazione in modo che queste possano variare indipendentemente.

  Ad esempio, nel progettare un'interfaccia grafica portabile, poniamo di volere un'astrazione Window che funzioni sia su X Window System che su IBM Presentation Manager (PM). Un approccio basato su ereditarietà potrebbe definire una classe astratta Window e due sottoclassi, XWindow e PMWindow, per i diversi sistemi. Questo approccio presenta però 2 problemi:

  – Risulta difficile estendere l'astrazione Window per supportare nuovi tipi di finestre o nuove piattaforme;

  – Per ogni nuova combinazione (es. IconWindow per finestre con icone) e piattaforma, bisogna creare più classi specifiche (XIconWindow, PMIconWindow, ecc.). Ogni nuova piattaforma moltiplica il numero di classi necessarie.

  Vediamo quindi di usare un approccio alternativo, dove una sola classe può sfruttare più *implementor*, cioè interfacce che espongono funzioni di basso livello da ciascuna delle API:

```
// impementor astratto
interface DrawingAPI {
    void drawWindow();
}

// implementor concreti

class XDrawingAPI {
    public void drawWindow() {
      // ...
    }
}

class IBMDrawingAPI {
    public void drawWindow() {
      // ...
    }
}

// classe vera e propria
class Window {
    private DrawingAPI api;

    public Window(DrawingApi api) {
      this.api = api;
    }

    public void draw() {
```

```
29        // basso livello
30        api.drawWindow();
31      }
32
33      public void resize() {
34        // alto livello
35        // ...
36      }
37  }
```

A questo punto sarà semplice creare oggetti *Window* che si appoggiano all'una o l'altra API, come segue:

```
1  Window XWindow = new Window(new XDrawingAPI());
2  Window IBMWindow = new Window(new IBMDrawingAPI());
```

- **Composite**: l'idea è di comporre oggetti in strutture ad albero che rappresentano gerarchie parti-complesso. In questo modo si possono trattare collezioni di oggetti, o loro singole istanze, in maniera uniforme. La chiave di questo pattern è quindi una classe astratta che rappresenta sia degli oggetti *primitivi*, che dei *container* di tali oggetti.

  L'esempio più semplice è quello di un file system, dove sia i file (*foglie* o *oggetti primitivi*) che le directory (*compositi* o *collezioni di oggetti*) sono istanze di una sola classe *FSNode*:

```
1  // interfaccia base di file e directory
2  interface FSNode {
3    String getName();
4    int getSize(); // in KB
5    void print(String indent);
6
7    // operazioni di composizione: default "non supportate" per le
       foglie
8    default void add(FSNode node) {
9      throw new UnsupportedOperationException(
10         getClass().getSimpleName() + " is a leaf");
11   }
12   default void remove(FSNode node) {
13     throw new UnsupportedOperationException(
14         getClass().getSimpleName() + " is a leaf");
15   }
16 }
17
18 // leaf (file)
19 class FileLeaf implements FSNode {
20   private final String name;
21   private final int sizeKB;
22
23   FileLeaf(String name, int sizeKB) {
24     this.name = name;
25     this.sizeKB = sizeKB;
26   }
27   public String getName() {
28     return name;
29   }
30   public int getSize() {
31     return sizeKB;
32   }
33   public void print(String indent) {
```

```
    System.out.printf("%s- %s (%d KB)%n", indent, name, sizeKB);
  }
}

// composite (directory)
class Directory implements FSNode {
  private final String name;
  private final List<FSNode> children = new ArrayList<>();

  Directory(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public int getSize() {
    // somma delle dimensioni dei figli
    return children.stream().mapToInt((e) -> e.getSize()).sum();
  }
  public void add(FSNode node) {
    children.add(node);
  }
  public void remove(FSNode node) {
    children.remove(node);
  }
  public void print(String indent) {
    System.out.printf("%s+ %s/ (%d KB)%n", indent, name, getSize());
    String childIndent = indent + " ";
    for (FSNode c : children) {
      c.print(childIndent);
    }
  }
}
```

- **Decorator**: questo è un pattern che si usa per aggiungere funzionalità, in maniera dinamica, a oggetti preesistenti. Forniscono quindi un alternativa alle sottoclassi per la gestione di funzionalità aggiuntive, quando vogliamo dare più responsabilità a singoli oggetti, e non ad intere classi.

  Torniamo all'esempio delle interfacce grafiche. Magari il nostro toolkit vuole fornire la possibilità di rendere qualsiasi componente *cliccabile* o *scrollabile*.

  Una soluzione semplice potrebbe essere quella di creare classi figlie di ogni componente, scrollabili e cliccabili.

  Un'alternativa è quella di *"avvolgere"* gli oggetti componente in altri oggetti, detti decoratori. I decoratori si conformano all'interfaccia degli oggetti che decorano, così da essere trasparenti agli utenti della classe.

  Vediamo quindi un esempio:

```
// componente base
interface Text {
  String render();
}

// implementazione concreta
class PlainText implements Text {
  private final String content;

```

```
  PlainText(String content) {
    this.content = content;
  }
  public String render() {
    return content;
  }
}

// classe base del decoratore, stessa interfaccia
abstract class TextDecorator implements Text {
  protected final Text inner;

  TextDecorator(Text inner) {
    this.inner = inner;
  }

  // possiamo ridefinire questo metodo!
  public abstract String render();
}

// decoratori concreti

class BoldDecorator extends TextDecorator {
  BoldDecorator(Text inner) {
    super(inner);
  }
  public String render() {
    return "<b>" + inner.render() + "</b>";
  }
}

class ItalicDecorator extends TextDecorator {
  ItalicDecorator(Text inner) {
    super(inner);
  }
  public String render() {
    return "<i>" + inner.render() + "</i>";
  }
}
```

- **Facade**: l'intento è di fornire un interfaccia unificata ad un insieme di interfacce in un sottosistema. In breve, una facade fornisce un interfaccia di livello più alto che rende l'intero sottosistema più semplice da usare.

  Prendiamo l'esempio di un emulatore, che contiene più componenti:

```
// componente astratto
abstract class SimulationComponent {
  void simulate();
}

// componenti concreti

class Bus extends SimulationComponent {
  // ...
}
class Memory extends SimulationComponent {
  // ...
}
class Processor extends SimulationComponent {
  // ...
```

```
}

// simulazione concreta, rappresenta la facade del sistema
class Simulation {
  Bus bus;
  Memory mem;
  Processor proc;

  public Simulation(Bus bus, Memory mem, Processor proc) {
    this.bus = bus;
    this.mem = mem;
    this.proc = proc;
  }

  public void step() {
    // la facade nasconde la complessita'
    bus.step();
    mem.step();
    proc.step();
  }
}
```

- **Flyweight**: utile quando abbiamo un grande numero di oggetti che condividono caratteristiche immutabili: decidiamo di utilizzare la condivisione delle risorse per risparmiare memoria. Sostanzialmente, quindi, l'approccio è simile a quello del *pooling*.

  Vediamo ad esempio un sistema che gestisce più caselle, che possono condividere lo stesso tipo:

```
// stato condiviso
record TileType(String texture, int movementCost) { }

// fabbrica di flyweight
class TileTypeFactory {
    private static final Map<String, TileType> cache = new HashMap
    <>();

    static TileType get(String texture, int cost) {
        return cache.computeIfAbsent(texture, k -> new TileType(
    texture, cost));
    }
}

// stato univoco: ottieniamo TileType condiviso dalla fabbrica
record Tile(int x, int y, TileType type) { }
```

- **Proxy**: qui vogliamo fornire un oggetto "fittizio", che fa da segnaposto per un altro oggeto. Il proxy può svolgere alcune funzioni:

  – Permettere di raggiungere oggetti in altri spazi di indirizzamento;
  – Controllare gli accessi agli oggett che rappresenta;
  – Rappresentare un oggetto costoso e crearlo solo quando necessario (*lazy instantiation* o *lazy loading* (se ci sono accessi al disco)).

  Un esempio di applicazione di questo pattern, tornando all'esempio del rendering dei documenti, è per gestire una classe *Image*, che va istanziata solo quando è effettivamente necessario disegnarla, ed è nel frattempo sostituita da un proxy.

```java
// interfaccia comune
interface Image {
  void display();
}

// classe reale (costosa da caricare)
class RealImage implements Image {
  private final String filename;

  public RealImage(String filename) {
    this.filename = filename;
    loadFromDisk();
  }
  private void loadFromDisk() {
    System.out.println("Loading image from disk: " + filename);
  }
  public void display() {
    System.out.println("Displaying image: " + filename);
  }
}

// classe proxy
class ImageProxy implements Image {
  private final String filename;
  private RealImage realImage; // inizialmente null

  public ImageProxy(String filename) {
    this.filename = filename;
  }
  public void display() {
    if (realImage == null) { // lazy loading
      realImage = new RealImage(filename);
    }
    realImage.display();
  }
}
```

### 12.0.2 Pattern comportamentali

Come ultima categoria di design pattern, parliamo dei pattern **comportamentali**. Questi riguardano il modo con cui gli oggetti interagiscono e si scambiano informazioni.

Sono, in particolare:

- **Chain of responsibility**: consiste nell'evitare di accoppiare la responsabilità di una richiesta al ricevitore di una richiesta, dando a più di un oggetto la possibilità di gestire tale richiesta. Quando arriva una richiesta, quindi, ogni gestore decide cosa fare e se necessario propagare la richiesta a altri gestori.

  Vediamo ad esempio un videogioco dove un giocatore può subire danno, e questo danno può essere elaborato sulla base di più fattori (difficoltà del gioco, presenza o meno di armatura, ecc...)

```java
class DamageRequest {
  public double amount;
  DamageRequest(double amount) {
    this.amount = amount;
  }
}

```

```
// handler astratto
abstract class DamageHandler {
  protected DamageHandler next;
  public DamageHandler setNext(DamageHandler next) {
    this.next = next;
    return next;
  }
  public void handle(DamageRequest req) {
    if (!process(req))
      return; // se false, la catena si ferma
    if (next != null)
      next.handle(req);
  }
  // true se deve continuare la catena
  protected abstract boolean process(DamageRequest req);
}

class ArmorHandler extends DamageHandler {
  private final double armor; // riduce di un certo numero di punti
    il danno
  ArmorHandler(double armor) {
    this.armor = armor;
  }
  protected boolean process(DamageRequest req) {
    double before = req.amount;
    req.amount = Math.max(0, req.amount - armor);
    System.out.println("[Armor] " + before + " -> " + req.amount);
    return req.amount > 0; // se il danno e' 0, fermo la catena
  }
}

class DifficultyHandler extends DamageHandler {
  private final double difficultyMultiplier;
  DifficultyHandler(double difficultyMultiplier) {
    this.difficultyMultiplier = difficultyMultiplier;
  }
  protected boolean process(DamageRequest req) {
    double before = req.amount;
    req.amount = req.amount * difficultyMultiplier;
    System.out.println("[Diff] " + before + " -> " + req.amount);
    return true;
  }
}

class Player {
  private double hp = 100;
  public void applyDamage(double dmg) {
    hp -= dmg;
    System.out.println(">> Player danneggiato " + dmg + ", HP = " +
    hp);
  }
}

public class Example {
  public static void main(String[] args) {
    Player player = new Player();
    // costruisco la catena: armor -> difficulty
    DamageHandler chain = new ArmorHandler(5);
    chain.setNext(new DifficultyHandler(1.5));
    DamageRequest req = new DamageRequest(30);
```

```
66      System.out.println("Danno base: " + req.amount);
67      chain.handle(req);
68      player.applyDamage(req.amount);
69    }
```

- **Command**: qui vogliamo incapsulare richieste all'interno di oggetti, permettendo di parametrizzare le richieste, metterle in code, e renderle annullabili.

  Il comando incapsula l'operazione da compiere, e su chi operare (il ricevitore), ma non l'implementazione dell'operazione da compiere (quella starà nel ricevitore).

  Vediamo ad esempio il classico gestionale bancario, dove prevediamo comandi di *Deposit* e *Withdraw* che operano su un certo oggetto ricevitore *BankAccount*:

```
1  interface Command {
2    void execute();
3    void undo();
4  }
5
6  // oggetto ricevitore
7  class BankAccount {
8    private int balance = 0;
9
10   public void deposit(int amount) {
11     balance += amount;
12     System.out.println("Deposit " + amount + " => balance = " +
       balance);
13   }
14   public void withdraw(int amount) {
15     balance -= amount;
16     System.out.println("Withdraw " + amount + " => balance = " +
       balance);
17   }
18   public int getBalance() {
19     return balance;
20   }
21 }
22
23 class DepositCommand implements Command {
24   private final BankAccount account;
25   private final int amount;
26
27   public DepositCommand(BankAccount account, int amount) {
28     this.account = account;
29     this.amount = amount;
30   }
31   public void execute() {
32     account.deposit(amount);
33   }
34   public void undo() {
35     account.withdraw(amount);
36   }
37 }
38
39 class WithdrawCommand implements Command {
40   private final BankAccount account;
41   private final int amount;
42
43   public WithdrawCommand(BankAccount account, int amount) {
44     this.account = account;
45     this.amount = amount;
```

```
  }
  public void execute() {
    account.withdraw(amount);
  }
  public void undo() {
    account.deposit(amount);
  }
}

class CommandManager {
  // mettiamo i comandi in una pila
  private final Stack<Command> history = new Stack<>();

  public void run(Command cmd) {
    cmd.execute();
    history.push(cmd);
  }
  public void undo() {
    if (history.isEmpty()) {
      System.out.println("Nothing to undo");
      return;
    }
    Command last = history.pop();
    last.undo();
  }
}

public class Example {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    CommandManager manager = new CommandManager();
    manager.run(new DepositCommand(account, 100));
    manager.run(new WithdrawCommand(account, 40));
    manager.run(new DepositCommand(account, 50));
    System.out.println(">> Undo last operation");
    manager.undo();
    System.out.println(">> Undo another operation");
    manager.undo();
    System.out.println("Final balance = " + account.getBalance());
  }
}
```

- **Interpreter**: un interprete per un linguaggio permette di definire delle **DSL** (*Domain Specific Language*), e avere un modo per interpretarle.

  L'implementazione degli *interpreter* è simile a quella dei *composite*, in quanto il modo più naturale per gestire grammatiche e attraverso strutture ad albero di dati nodi sintattici. La differenza principale dal composite è però che si definisce, appunto, una *grammatica*, cioè una struttura specifica che l'albero che formiamo deve assumere.

  In un linguaggio come il Java, scrivere interpreti risulta molto semplice:

```
// espressione astratta
interface Expression {
  int interpret();
}
// espressione terminale (numero)
class Number implements Expression {
  private final int value;
```

```
8
9   public Number(int value) {
10    this.value = value;
11  }
12  public int interpret() { return value; }
13}
14
15// espressioni non terminali: somma o sottrazione
16
17class Plus implements Expression {
18  private final Expression left, right;
19
20  public Plus(Expression left, Expression right) {
21    this.left = left;
22    this.right = right;
23  }
24  public int interpret() {
25    return left.interpret() + right.interpret();
26  }
27}
28
29class Minus implements Expression {
30  private final Expression left, right;
31
32  public Minus(Expression left, Expression right) {
33    this.left = left;
34    this.right = right;
35  }
36  public int interpret() {
37    return left.interpret() - right.interpret();
38  }
39}
40
41// ...
```

- **Iterator**: gli iteratori, come già visti in C++, forniscono un modo per accedere agli elementi di un oggetto aggregato in maniera sequenziale, mascherando all'utente dell'iteratore il funzionamento di tale attraversamento sequenziale (e quindi fornendo un interfaccia unificata).

```
1interface MyIterator<T> {
2  boolean hasNext();
3  T next();
4}
5
6class MyCollection<T> {
7  private List<T> l = new ArrayList<>();
8
9  public void add(T elem) {
10   l.add(elem);
11 }
12 public MyIterator<T> iterator() {
13   return new MyListIterator<>(l);
14 }
15}
16
17class MyListIterator<T> implements MyIterator<T> {
18  private List<T> list;
19  private int pos = 0;
20
```

```
  public MyListIterator(List<T> list) {
    this.list = list;
  }
  @Override
  public boolean hasNext() {
    return pos < list.size();
  }
  @Override
  public T next() {
    if (!hasNext()) {
      throw new IllegalStateException("Non ci sono piu' elementi");
    }
    return list.get(pos++);
  }
}
```

- **Mediator**: è un pattern dove gestiamo più oggetti che comunicano fra di loro attraverso un singolo oggetto mediatore. Permette di gestire un insieme di oggetti con molte relazioni oggetto-oggetto.

  In questo caso, anziché avere molte relazioni, ogni oggetto comunica solo con il mediatore. Riduce accoppiamento e semplifica manutenzione e evoluzione. La logica di comunicazione diventa centralizzata. Ad esempio, in una GUI, se un campo viene selezionato se ne deselezionano altri e così via...

  Vediamo l'esempio di una chat room dove più utenti si scambiano messaggi comunicando con lo stesso mediatore:

```
interface ChatMediator {
  void sendMessage(String msg, User sender);
  void addUser(User user);
}

class ChatRoom implements ChatMediator {
  private final List<User> users = new ArrayList<>();
  @Override
  public void addUser(User user) {
    users.add(user);
  }
  @Override
  public void sendMessage(String msg, User sender) {
    for (User u : users) {
      if (u != sender) {
        u.receive(msg, sender.getName());
      }
    }
  }
}
abstract class User {
  protected ChatMediator mediator;
  protected String name;
  public User(ChatMediator mediator, String name) {
    this.mediator = mediator;
    this.name = name;
    mediator.addUser(this);
  }
  public String getName() {
    return name;
  }
  public abstract void send(String msg);
```

```
  public abstract void receive(String msg, String from);
}
lass ChatUser extends User {
  public ChatUser(ChatMediator mediator, String name) {
    super(mediator, name);
  }
  @Override
  public void send(String msg) {
    System.out.println(name + " sends: " + msg);
    mediator.sendMessage(msg, this);
  }
  @Override
  public void receive(String msg, String from) {
    System.out.println(name + " receives from " + from + ": " + msg);
  }
}
public class Example {
  public static void main(String[] args) {
    ChatMediator mediator = new ChatRoom();
    User mario = new ChatUser(mediator, "Mario");
    User luigi = new ChatUser(mediator, "Luigi");
    User peach = new ChatUser(mediator, "Peach");
    mario.send("Ciao a tutti!");
    luigi.send("Ciao!");
    // ....
  }
}
```

- **Memento**: il memento è un pattern secondo il quale immagazziniamo lo stato (immutabile) di un oggetto per poterlo ricaricare in un istante successivo. Il *memento* è quindi l'oggetto immutabile che mantiene lo stato, mentre un'altro oggetto (detto *caretaker*) sarà quello che si occuperà di prelevare, memorizzare e restituire successivamente lo stato.

  Poniamo l'esempio di un sistema di salvataggio in un videogioco:

```
// memento
class PlayerState {
  private final int life;
  private final int x, y;
  private final List<String> inventory;

  public PlayerState(int life, int x, int y, List<String> inventory)
    {
    this.life = life;
    this.x = x;
    this.y = y;
    // Crea una copia della lista
    this.inventory = new ArrayList<>(inventory);
  }
  public int getLife() {
    return life;
  }
  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  public List<String> getInventory() {
```

```
    // restituisce una copia non modificabile
    return Collections.unmodifiableList(inventory);
  }
}

// oggetto originatore dei dati
class Player {
  private int life = 100;
  private int x = 0, y = 0;
  private final List<String> inventory = new ArrayList<>();

  public void move(int dx, int dy) {
    x += dx;
    y += dy;
  }
  public void damage(int amount) {
    life -= amount;
  }
  public void addItem(String item) {
    inventory.add(item);
  }
  public void printStatus() {
    System.out.println(
        "Life=" + life + ", pos=(" + x + "," + y + "), items=" +
     inventory);
  }
  // crea un memento dallo stato
  public PlayerState saveState() {
    return new PlayerState(life, x, y, inventory);
  }
  // ripristina lo stato da un memento
  public void restoreState(PlayerState m) {
    this.life = m.getLife();
    this.x = m.getX();
    this.y = m.getY();
    inventory.clear();
    // Li riaggiunge tutti
    inventory.addAll(m.getInventory());
  }
}

// caretaker
class SaveManager {
  private PlayerState checkpoint;

  public void save(PlayerState m) {
    checkpoint = m;
  }
  public PlayerState load() {
    return checkpoint;
  }
}
```

A questo punto salvataggio e caricamento dello stato del giocatore sono banali come:

```
Player player = new Player();
SaveManager saver = new SaveManager();

saver.save(player.playerState()); // salva

```

```
// ...

player.restoreState(saver.load()); // carica
```

- **Observer**: questo pattern permette di realizzare un sistema di *eventi*, anche detto un sistema *publish-subscribe*, dove un certo oggetto (detto *osservatore*) sottoscrive agli eventi generati da altri oggetti (che vengono detti *osservati*).

  Questo può essere utile nel caso di oggetti che devono propagare cambiamenti di stato verso l'alto, senza necessariamente chi è interesato ad elaborare tali cambiamenti.

  Un esempio può essere quello dell'emulatore visto per le *facade*: poniamo infatti che ogni componente sia capace di generare un evento, e che la nostra facade voglia rilevare tali eventi:

```
// evento
class SimulationEvent {
  private final String mess;

  public String getMess() {
    return mess;
  }
}

interface SimulationListener {
  void onEvent(SimulationEvent event);
}

// componente astratto
abstract class SimulationComponent {
  private List<SimulationListener> listeners = new ArrayList<>();

  void subscribe(SimulationListener listener) {
    listeners.add(listener);
  }

  void simulate();
  void raiseEvent(SimulationEvent event) {
    for(SimulationListener listener : listeners) {
      listener.onEvent(event);
    }
  }
}

// componenti concreti

class Bus extends SimulationComponent {
  // ...
}
class Memory extends SimulationComponent {
  // ...
}
class Processor extends SimulationComponent {
  // ...
}

// simulazione concreta, rappresenta la facade del sistema
class Simulation implements SimulationListener {
  Bus bus;
```

```
  Memory mem;
  Processor proc;

  public Simulation(Bus bus, Memory mem, Processor proc) {
    this.bus = bus;
    this.mem = mem;
    this.proc = proc;
  }

  public void init() {
    bus.subscribe(this);
    mem.subscribe(this);
    proc.subscribe(this);
  }

  public void onEvent(SimulationEvent e) {
    System.out.println(e.getMess());
  }

  public void step() {
    // la facade nasconde la complessita'
    bus.step();
    mem.step();
    proc.step();
  }
}
```

- **State**: vogliamo incapsulare lo *stato* di un oggetto all'interno di classi, e gestire tali stati attraverso un *contesto* comune. Le chiamate al contesto verrano redirette allo stato corrente, e lo stato corrente avrà un riferimento al contesto per poter aggiornare lo stato.

  Facciamo l'esempio di un riproduttore musicale che ha 2 stati: lo stato in *pausa* e lo stato in *riproduzione* di musica:

```
// interfaccia di stato
interface PlayerState {
  void pressButton(MusicPlayer player);
}

// stati concreti
class PlayingState implements PlayerState {
  @Override
  public void pressButton(MusicPlayer player) {
    System.out.println("Pausing music...");
    player.setState(new PausedState());
  }
}

class PausedState implements PlayerState {
  @Override
  public void pressButton(MusicPlayer player) {
    System.out.println("Playing music...");
    player.setState(new PlayingState());
  }
}

// contesto
class MusicPlayer {
  private PlayerState state = new PausedState(); // stato iniziale
```

```
26    public void setState(PlayerState state) {
27      this.state = state;
28    }
29    public void pressButton() {
30      state.pressButton(this);
31    }
32  }
33
34  public class Esempio {
35    public static void main(String[] args) {
36      MusicPlayer player = new MusicPlayer();
37      player.pressButton(); // Playing
38      player.pressButton(); // Pausing
39      player.pressButton(); // Playing
40    }
41  }
```

- **Strategy**: questo pattern consiste nel prendere famiglie di algoritmi (algoritmi con gli stessi argomenti che danno risultati dello stesso tipo, quindi rappresentabili da interfacce), ed incapsularli per renderli intercambiabili.

  Gli oggetti che useranno gli algoritmi potranno quindi vedere la strategia usata cambiare nel tempo sulla base di diverse implementazioni.

  Poniamo l'esempio di un validatore di password che può usare diversi algoritmi di validazione:

```
1  // interfaccia di strategia
2  interface PasswordStrategy {
3    boolean isValid(String password);
4  }
5
6  // strategie concrete
7
8  class SimplePassword implements PasswordStrategy {
9    public boolean isValid(String pwd) {
10       // piu' di 5 caratteri
11       // ...
12   }
13 }
14
15 class StrongPassword implements PasswordStrategy {
16   public boolean isValid(String pwd) {
17       // piu' di 7 caratteri
18       // ....
19   }
20 }
21
22 // contesto delle strategie
23 class PasswordValidator {
24   private PasswordStrategy strategy;
25   public void setStrategy(PasswordStrategy strategy) {
26     this.strategy = strategy;
27   }
28   public void validate(String pwd) {
29     if (strategy == null) {
30       System.out.println("Nessuna strategia impostata!");
31       return;
32     }
33     System.out.println(
```

```
        "Password \"" + pwd + "\" valida? " + strategy.isValid(pwd));
  }
}

public class Example {
  public static void main(String[] args) {
    PasswordValidator validator = new PasswordValidator();
    validator.setStrategy(new SimplePassword());
    validator.validate("ciao12"); // true
    validator.setStrategy(new StrongPassword());
    validator.validate("ciao12"); // false
    validator.validate("Ciao1234"); // true
  }
}
```

- **Template method**: l'idea è di definire lo *"scheletro"* di un algoritmo che definisce un operazione, mantenendo astratti i passaggi dell'esecuzione di tale algoritmo. In seguito, si potrà specializzare la classe, riempiendo i passaggi astratti con implementazioni concrete.

  Vediamo un esempio:

```
// template method
abstract class Processor {
  public final void() {
    step0();
    step1();
    step2();
  }

  protected abstract void step0();
  protected abstract void step1();
  protected abstract void step2();
}

// implementazioni concrete

class ProcessorA {
  protected void step0() {
    // ...
  }
  protected void step1() {
    // ...
  }
  protected void step2() {
    // ...
  }
}

class ProcessorB {
  protected void step0() {
    // ...
  }
  protected void step1() {
    // ...
  }
  protected void step2() {
    // ...
  }
}
```

- **Visitor**: fornisce un alternativa alla overriding di metodi astratti. Si prevede infatti un a certa classe, detta *visitor*, che prevede più metodi, ognuno capace di gestire un istanza di classe in una certa gerarchia di classi.

  A questo punto le classi nella gerarchia dovranno solo implementare un metodo per l'accettazione del visitor.

  Vediamo l'esempio banale, che è quello di un visitor per il calcolo delle aree di diversi specializzazioni di una classe *Shape*:

```
// elemento astratto
interface Shape {
  void accept(ShapeVisitor visitor);
}

// elementi concreti
class Circle implements Shape {
  double radius;

  public Circle(double radius) {
    this.radius = radius;
  }
  public double getRadius() {
    return radius;
  }

  // questo va ridefinito qui (!)
  // non possiamo usare this nei metodi default
  @Override
  public void accept(ShapeVisitor visitor) {
    visitor.visit(this);
  }
}

class Rectangle implements Shape {
  double w, h;

  public Rectangle(double w, double h) {
    this.w = w;
    this.h = h;
  }
  public double getWidth() {
    return w;
  }
  public double getHeight() {
    return h;
  }

  // come sopra
  @Override
  public void accept(ShapeVisitor visitor) {
    visitor.visit(this);
  }
}

// visitor
class AreaVisitor {
  public void visit(Circle c) {
    double area = Math.PI * c.getRadius() * c.getRadius();
    System.out.println("Area cerchio: " + area);
  }
```

```
52
53    public void visit(Rectangle r) {
54      double area = r.getWidth() * r.getHeight();
55      System.out.println("Area rettangolo: " + area);
56    }
57  }
```

A questo punto si può calcolare l'area di una qualsiasi forma con:

```
Shape rect = new Rectangle(2, 5);
Shape circ = new Circle(3);

AreaVisitor visitor = new AreaVisitor();
rect.accept(visitor); // area rettangolo
circ.accept(visitor); // area cerchio
```

Questo è utile, in quanto l'alternativa basata sull'ereditarietà prevedeva la definizione di un qualche metodo:

```
public double calcola(Shape s) {
  if (s instanceof Circle) {
    Circle c = (Circle) s;
    return Math.PI * c.getRadius() * c.getRadius();
  } else if (s instanceof Rectangle) {
    Rectangle r = (Rectangle) s;
    return r.getWidth() * r.getHeight();
  }
  throw new IllegalArgumentException("Forma sconosciuta");
}
```

che è molto brutto in quanto:

- Presenta molti `if-else`;
- Richiede il costrutto `instanceof`, e quindi il controllo dinamico del tipo di classe (con riflessione).

Il nostro approccio è più compatto e rimanda la discriminazione fra classi ad un meccanismo base del linguaggio (l'overriding delle funzioni con stesso nome ma diversi argomenti).

Notiamo infine che un altro vantaggio è quello che si possono definire più visitatori (magari da un interfaccia comune), che si occupano di implementare operazioni diverse sugli oggetti che visitano (senza dover modificare gli oggetti stessi).

Notiamo che un pattern che è emerso finora è quello del *programming against interfaces*: quello che vogliamo in genere fare è definire prima le interfacce, e poi le classi concrete che implementano tali interfacce.

In questo modo, il codice degli utenti delle classi non dovrà cambiare: piuttosto, a cambiare sarà l'istanziazione delle classi concrete vere e proprie.