1 Lezione del 06-11-25

Continuiamo con la discussione di costrutti avanzati del linguaggio Java.

1.0.1 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.

1.0.2 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.

1.1 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.

1.2 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).

1.2.1 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*).

1.2.2 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*.

1.3 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.

1.3.1 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:

```
1 public class Logger {
2   private Logger() {
3     // qui si inizializza (1 volta) il logger
4   }
5
```

```
  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**:
- **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);
  }
}
```

- **Prototype**: