Programmazione Orientata agli Oggetti

Elementi di programmazione multi-thread in Java
Concetti di Lambda Expression e Stream
Alcune novità delle ultime versioni di Java

Libero Nigro

Concetti di base

- I moderni sistemi operativi (Windows, Linux, MacOS, etc.) sono multi-tasking, nel senso che consentono a più programmi di essere eseguiti contemporaneamente.
- Nel caso di una singola CPU (mono-core), il meccanismo sottostante al multi-tasking è il *time-slicing* (o a divisione di tempo, o con *quanto* di tempo). Trascorso il quanto (es. 10ms), anche se il programma non ha finito, viene estromesso (*pre-emption*) dalla CPU e un altro programma viene caricato per eseguire il suo quanto (*context-switch*).
- In realtà, pure nel caso di CPU multi-core, di norma il numero dei programmi da eseguire simultaneamente è superiore al numero dei core disponibili, per cui sussistono sempre i problemi di pre-emption e context-switch su ogni core
- Un programma in esecuzione si accompagna a strutture dati di supporto (es. per memorizzare il contenuto dei registri al tempo di un context-switch) per cui è più opportunamente riferito col nome di *processo.*
- Un'intera applicazione Java diventa un processo.
- Java permette di strutturare un programma/processo in più *thread* o unità concorrenti. I thread sono un livello più fine dei processi, ma i concetti di pre-emption e context-switch sono molto simili
- Importante: un thread (o un processo) non sa quando può capitare la pre-emption. Tutto questo (*non-determinismo*) complica la programmazione concorrente, per la necessità di adottare provvedimenti (es. lucchetti) per evitare interferenze/ inconsistenze sui dati e dunque impredicibilità e malfunzionamenti.

Programmazione multi-thread in Java

- La classe base **Thread** (non abstract) e l'interfaccia **Runnable** (col solo metodo astratto *public void run()*) consentono di introdurre *classi di oggetti thread*
- I metodi della classe Thread includono: *start()*, *destroy(), setName(String), getName(), yield(), getPriority(), setPriority(p), interrupt(), isInterrupted()*, etc. e metodi statici come *sleep(durata in ms)*, etc.
- Il comportamento di un oggetto thread è specificato dal suo metodo run() (senza parametri) invocato dalla JVM
- Per dare un'idea della programmazione concorrente in Java, di seguito si mostra un semplice programma in cui due processi (thread) A e B devono accedere in mutua esclusione ad una risorsa condivisa, nel rispetto dei seguenti vincoli (*protocollo*): se A e B si alternano, il protocollo ritorna sempre al suo stato di riposo. Se però A accede due volte consecutivamente alla risorsa, al terzo tentativo di accesso, A verrà bloccato sino a che B non faccia un suo accesso.
- Il protocollo mira ad evitare la *starvation*, ossia la possibilità che un processo possa aspettare un tempo illimitato prima di accedere alla risorsa. Un altro problema che affligge in generale i sistemi concorrenti è il *deadlock*, una situazione di blocco «fatale» nella quale i processi si bloccano in attesa di un'operazione di un altro processo, che però mai arriva.

La classe Processo

```
package poo.thread;
public class Processo extends Thread {
    public enum Proc{A,B}
    private Proc id;
    private int MAX=5000, MIN=1000;
    private Manager m;
    public Processo( Proc id, Manager m ) { this.id=id; this.m=m; }
    private void pausa() {
        try {
            Thread.sleep( (int)(Math.random()*(MAX-MIN)+MIN));
        }catch( InterruptedException ie ) {}
    }//pausa
    public void run() {
        while( true ) {
            pausa();
            m.richiesta(id);
            pausa();
            m.rilascio(id);
        }
    }//run
}//Processo
```

La classe thread-safe Manager

```
package poo.thread;
public class Manager {
    private int count=0; //conta accessi consecutivi di A
    private boolean risorsaOccupata=false;
    public synchronized void richiesta( Processo.Proc id ) {
        if( id==Processo.Proc.A ) {
            System.out.println("Processo A fa richiesta.");
            while( risorsaOccupata || count==2 ) {
                try { wait(); }catch( InterruptedException ie ) {}
            }//while
            count++;
            System.out.println("Processo A ottiene risorsa.");
            risorsaOccupata=true;
        }
        else { //B
            System.out.println("Processo B fa richiesta.");
            while( risorsaOccupata ) {
                try { wait(); }catch( InterruptedException ie ) {}
            }//while
            System.out.println("Processo B ottiene risorsa.");
            risorsaOccupata=true;
        }
    }//richiesta
```

```
    public synchronized void rilascio( Processo.Proc id ) {
        if( id==Processo.Proc.B ) count=0;
        System.out.println("Processo "+id+" rilascia la risorsa.");
        risorsaOccupata=false;
        notifyAll();
    }//rilascio

}//Manager
--------------------------------------------------------------------------------
package poo.thread;

public class Main {
    public static void main( String[] args ) {
        Manager m=new Manager();
        Processo a=new Processo( Processo.Proc.A, m );
        Processo b=new Processo( Processo.Proc.B, m );
        a.start(); b.start();
}   //main
}//Main
```

Il monitor nativo di Java in breve…

- Ogni oggetto Java è provvisto di un *lucchetto*, che nasce aperto. Al lucchetto è associato un dormitorio, detto *wait-set*, NON FIFO.
- L'ingresso in un metodo sincronizzato è possibile solo se il lucchetto di this è aperto, nel qualcaso viene immediatamente chiuso.
- Se il lucchetto è chiuso, il thread che invoca il metodo sincronizzato è posto a dormire sul wait-set.
- Non appena si esce da un metodo sincronizzato, ***un*** thread in attesa (tra quelli che avevano trovato il lucchetto chiuso) può essere svegliato, e può impossessarsi del lucchetto ed eseguire il metodo sincronizzato.
- I metodi *wait(), notify() e notifyAll()* (sempre meglio notifyAll()) sono esportati da Object. Il metodo wait() mette esplicitamente a dormire il thread sul wait-set, e riapre il lucchetto. notifyAll() sveglia tutti i thread che hanno eseguito wait sul wait-set ma ciascuno: a) dovrà riguadagnare il lucchetto; b) ritestare la condizione che, se non è vera, comporta il tornare a dormire rieseguendo wait. Questo (ma ci sono anche altre ragioni) spiega perché la wait() va *sempre* avviluppata in un ciclo di while come mostrato nell'esempio.
- **Attenzione**: notifyAll() non sveglia i thread in attesa sul wait-set perchè hanno trovato il lucchetto chiuso. Questi thread si svegliano, uno alla volta, automaticamente e implicitamente quando il lucchetto si riapre.

Esempio di run

Processo B fa richiesta.
Processo B ottiene risorsa.
Processo A fa richiesta.
Processo B rilascia la risorsa.
Processo A ottiene risorsa.
Processo B fa richiesta.
Processo A rilascia la risorsa.
Processo B ottiene risorsa.
Processo A fa richiesta.
Processo B rilascia la risorsa.
Processo A ottiene risorsa.
Processo B fa richiesta.
Processo A rilascia la risorsa.
Processo B ottiene risorsa.
Processo A fa richiesta.
Processo B rilascia la risorsa.
Processo A ottiene risorsa.
Processo B fa richiesta.
Processo A rilascia la risorsa.
Processo B ottiene risorsa.

Processo A fa richiesta.
Processo B rilascia la risorsa.
Processo A ottiene risorsa.
Processo A rilascia la risorsa.
Processo A fa richiesta.
Processo A ottiene risorsa.
Processo B fa richiesta.
Processo A rilascia la risorsa.
Processo B ottiene risorsa.
Processo B rilascia la risorsa.
Processo B fa richiesta.
Processo B ottiene risorsa.
Processo A fa richiesta.
Processo B rilascia la risorsa.
Processo A ottiene risorsa.
Processo A rilascia la risorsa.
Processo B fa richiesta.
Processo B ottiene risorsa.
…

Nella prima colonna i processi si alternano.
Nella seconda, in neretto, si mostra un caso allorquando A realizza due accessi consecutivi dopo cui solo B può accedere etc. Si suggerisce di testare il programma anche quando le costanti MAX e MIN sono poste a 0.

L’interfaccia funzionale Runnable

- Dispone di un solo metodo astratto:
  - public void run()

- Un oggetto Runnable, istanza di una classe che implementa Runnable, costituisce un task (un compito).
- Sia task un tale oggetto. Per creare e porre in esecuzione un thread dedicato a realizzare questo task, si procede come segue:

- Thread t=new Thread(task); //task è un oggetto Runnable
- t.start();

- All’atto pratico, anziché costruire una classe esplicita che implementa Runnable ed istanziarla per ottenere un oggetto task, si può fornire equivalentemente una lambda expression come indicato più avanti in questo capitolo.

Interfacce funzionali e lambda expression

- Si chiama *interfaccia funzionale (FI-Functional Interface)* una qualsiasi interfaccia che abbia **un solo** metodo astratto. Del metodo non interessa il nome ma l'ordine e i tipi dei parametri ed il tipo di risultato che ritorna.
- **Attne**: l'interfaccia potrebbe anche avere N metodi, di cui N-1 es. realizzati in veste default o static, ma uno soltanto deve essere astratto.
- In un qualsiasi punto di un programma dove è atteso un oggetto-istanza di tipo interfaccia funzionale, è possibile equivalentemente:
  - Fornire (anche al volo) una inner class (es. anonima) che implementa l'interfaccia, e usare un oggetto istanza di questa classe.
  - Specificare una lambda expression il cui corpo di esecuzione fornisce il codice del metodo astratto dell'interfaccia. La lambda può essere preferibile perché più compatta e meno verbosa rispetto a programmare una classe.
  - Specificare un **method reference** esistente.
- Una lambda expression è a tutti gli effetti un *oggetto funzionale*. Si può passare come parametro a, o restituire come risultato da, un metodo. Notare, tuttavia, che una lambda expression è anonima e non specifica il tipo del valore di ritorno.
- Nella prossima slide si mostrano semplici esempi di lambda expression. Di ciascun si dovrebbe intuire il tipo della FI cui la lambda si riferisce.
- Un qualunque metodo (static, non static, costruttore) di una classe che risponda ai requisiti del metodo astratto di una FI, può essere usato come reference method.

Esempi

(String s1, String s2) -> { return s1.length()-s2.length(); } //FI tipo Comparator

(s1,s2) -> {return s1.length()-s2.length(); } //il compilatore deduce il tipo di s1 e s2 dal contesto //type inference

(s1,s2) -> s1.compareTo(s2)

(int n) -> Math.sqrt(n) //FI tipo **Function<T,R>** accetta un dato di tipo T e ritorna un valore di tipo R.

s -> System.out.println(s)
//FI tipo **Consumer<T>** con un metodo che riceve un oggetto e non ritorna nulla.

s -> s.length()>=12 //FI con un metodo che riceve un oggetto e ritorna boolean (**Predicate**)

p -> p.getEta()>=18 && p.sesso()==Persona.Sex.Femmina (sempre **Predicate**)

Laddove è specificabile una lambda, si può anche fornire il riferimento a un metodo (statico o no) esportato da una classe, che abbia lo stesso mapping argomenti/risultato. Si tratta del concetto (versatile e compatto) di **method reference**.

Uso di FI, lambda e method reference

```
package poo.lambda;
public class Auxiliary {

    public static int method1( String s1, String s2 ) {
        if( s1.length()<s2.length() ||
            s1.length()==s2.length() && s1.compareTo(s2)<0 ) return -1;
        if( s1.equals(s2) ) return 0;
        return 1;
    }//method1

    public int method2( String s1, String s2 ) {
        if( s1.length()>s2.length() ||
            s1.length()==s2.length() && s1.compareTo(s2)<0 ) return -1;
        if( s1.equals(s2) ) return 0;
        return 1;
    }//method2

}//Auxiliary
```

```
package poo.lambda;
import java.util.Collections;
import java.util.List;

public class LambdaDemo {
    public static void main( String... args) {
    List<String> ls=java.util.Arrays.asList("casa","zaino","duca","cuore","albero","oro","dado");
    System.out.println( ls ); // situazione iniziale (0)
    Collections.sort( ls, (s1,s2)->Integer.compare(s1.length(),s2.length()) ); //lambda expr (1)
    System.out.println( ls ); Collections.shuffle( ls );
    Collections.sort( ls, Auxiliary::method1 ); //static method reference (2)
    System.out.println( ls ); Collections.shuffle(ls);
    Auxiliary a=new Auxiliary();
    Collections.sort( ls, a::method2 ); //non static method reference, on explicit object (3)
    System.out.println( ls ); Collections.shuffle(ls);
    Collections.sort( ls, new Auxiliary()::method2 );  //non static meth ref, anonymous object (4)
    System.out.println( ls );
    }//main
}//LambdaDemo
```

Output

[casa, zaino, duca, cuore, albero, oro, dado] (0)
[oro, casa, duca, dado, zaino, cuore, albero] (1)
[oro, casa, dado, duca, cuore, zaino, albero] (2)
[albero, cuore, zaino, casa, dado, duca, oro] (3)
[albero, cuore, zaino, casa, dado, duca, oro] (4)

(0) riflette la situazione iniziale della lista
(1) corrisponde ad una lambda expr che confronta unicamente le lunghezze delle stringhe
(2) effetto di uno static method reference che confronta prima le length e poi l'ord alfabetico
(3) non static method reference, su oggetto esplicito
(4) non static method reference, su oggetto anonimo

Costruttori come method reference

- Se il metodo di una FI può essere riprodotto dal costruttore di una classe, Java consente di utilizzare questo costruttore come reference method.
- Ad esempio, considerato che la FI Runnable ha il metodo **void run(){}** che non riceve parametri e ritorna void, è possibile fare quanto segue:

```
public class Task {
    public Task() {
        System.out.println("Task invoked...");
    }//Task

    public static void main(String[] args) {
        Thread t=new Thread( Task::new );
        t.run();
    }//main
}//Task
```

La classe Task ha il solo costruttore di default. Il corpo di tale costruttore può essere usato come corpo del metodo run() di un Runnable, che in questo caso si riduce ad una stampa.
Creando un oggetto Thread a partire da un task, si può specificare una lambda expression implicita usando il costruttore di Task come reference method (notare la sintassi).

Un altro esempio

```
package poo.lambda;
@FunctionalInterface
interface String2Message{
    Message getMessage(String msg);
}//String2Message

class Message{
   public Message(String msg){//constructor
      System.out.print(msg);
   }
}//Message

public class ConstructorAsMethodReference {
     public static void main(String[] args){
          String2Message hello = Message::new;
          hello.getMessage("Hello there! Here I'm.");
     }//main
}//ConstructorAsMethodReference
```

Il costruttore della classe Message crea un Message da una String. Può quindi essere usato come method reference per la FI String2Message

Una finestra di cambio con lambda expr

```
package poo.swing;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class FinestraCambioFTL extends JFrame{
   private JTextField euro, lire;
   private final float EURO_LIRE=1936.27f;
   private ActionListener al= ( evt ) -> {
     if( evt.getSource()==euro ){
        String ae=euro.getText();
        if( !ae.matches("[0-9]+(\\.[0-9]+)?") ){
           euro.setText("WHAT?"); lire.setText("?");
           return;
        }
        double e=Double.parseDouble( ae );
        euro.setText( String.format("%1.2f",e) );
        double lit=Math.round(e*EURO_LIRE);
        lire.setText( String.format("%1.0f",lit) );
     }
else if( evt.getSource()==lire ){
     String al=lire.getText();
     if( !al.matches("[0-9]+") ){
        lire.setText("WHAT?");
        euro.setText("?");  return;
     }
      double eur=Double.parseDouble
        ( al )/EURO_LIRE;
      euro.setText( String.format("%1.2f",eur) );
    }
 }; //fine lambda expression
```

```
public FinestraCambioFTL(){
    setTitle("Cambio Euro-Lire");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel p=new JPanel();
    p.add( new JLabel("Euro", JLabel.RIGHT) );
    p.add( euro=new JTextField("",12) );
    p.add( new JLabel("Lire", JLabel.RIGHT) );
    p.add( lire=new JTextField("",12) );
    add(p, BorderLayout.NORTH);
    JPanel q=new JPanel();
    q.add( new JLabel("1 Euro = 1936.27 Lire", JLabel.RIGHT) );
    add( q, BorderLayout.SOUTH );
    euro.addActionListener( al );
    lire.addActionListener( al );
    setLocation(400,400);  setSize(450,100);
  }
}//FinestraCambioFTL

public class CambioFTL{
  public static void main(String []args){
    FinestraCambioFT fc=new FinestraCambioFT();  fc.setVisible(true);
  }
}//CambioFTL
```

Alcune FI della libreria di Java

- La libreria di Java è ricca di interfacce funzionali (si consulti ad es. il package *java.util.function*) in corrispondenza delle quali si possono specificare lambda expression o reference method. Alcune di queste standard FI sono:

  - **Consumer<T>** col metodo **void apply(T elem)**
  - **Predicate<T>** col metodo **boolean test(T elem)**
  - **Function<T,R>** con un metodo **R apply(T elem)**
  - **Supplier<T>**, con un metodo **T get()**, genera dati di tipo T
  - **BiFunction<T,U,R>**, con un metodo **R apply(T e1, U e2)**
  - ...

- Le FI possono essere (e spesso sono) generiche in uno o più parametri tipo.
- La definizione di una lambda expression, invece, non può mai essere generica.

Vincoli di visibilità delle lambda expression

- Una lambda è ovviamente programmata in un certo contesto e vede i dati dell'ambiente in cui è immersa.
- Stante il carattere dinamico del ciclo di vita di una lambda, può benissimo accadere che l'oggetto che definisce l'ambiente circostante può non esistere nel momento che la lambda viene eseguita.
- Per le ragioni di cui sopra, una lambda dovrebbe limitarsi ad usare solo dati *final* (costanti) dell'ambiente circostante, o cosiddetti «*effettivamente final*», ossia dati che pur essendo variabili, non vengono modificati dalla lambda.

Lambda expression e ricorsione

- Essendo una lambda expr molto simile ad un metodo, c'è da chiedersi se possa essere strutturata in versione ricorsiva. La risposta è sì, purchè si riesca a dar un nome alla lambda e si garantisca la restituzione del risultato. Segue un esempio.

```
public class Fattoriale {
  @FunctionalInterface
  interface Fact{
    long fact( int n );
  }
  public static void main( String[] args ) {
    Fact[] f=new Fact[1];
    f[0]=(int n) ->{if( n<=1 ) return 1;
                       return n*f[0].fact(n-1);
                   };
    int n=5;
    System.out.println( "fact("+n+")="+f[0].fact(n) );
  }//main
}//Fattoriale
```

Una lambda ricorsiva per
Il calcolo del fattoriale di un numero intero n.
L'array di un singolo elemento f[] fornisce «il nome» alla lambda.
La restituzione del risultato è garantita dalla definizione del corpo della lambda.

Il metodo forEach sulle collezioni (JDK >=8)

```
import java.util.ArrayList;
import java.util.List;
public class For_Each {
   public static void main( String... args) {
      List<String> ls=new ArrayList<>();
      ls.add("fuoco"); ls.add("banca"); ls.add("fieno"); ls.add("bordo");
      ls.add("aia"); ls.add("bio");

      //elaborazione con iteratore esterno
      for( String s: ls ) System.out.println(s);  //istruzione for-each

      //elaborazione col metodo forEach e interatore interno (o implicito)
      ls.forEach( s->System.out.println(s) ); //con lambda per l'oggetto Consumer
      ls.forEach( System.out::println ); //con reference method per il Consumer
   }
}//For_Each
```

Mentre le mappe non dispongono di iteratore, a partire da Java 8 esse dispongono comunque del metodo forEach che consente di applicare un'azione a tutte le coppie (k,v). La FI in gioco è **BiConsumer<T,U>** che riceve due parametri come K e V e applica un'azione.

```
Map<Integer, String> map = new HashMap<>();
map.put(10, "Lessie");
map.put( 20, "Tom" );
map.put( 30, "Faber" );

map.forEach(  (key, value) -> System.out.println(key + " " + value)  );

map.forEach(  (k,v)->{  if( k==30) System.out.println(v); }  ) ;
```

Nozioni di **stream** sulle collezioni e «fluent programming»

- Sulle collezioni di Java 8 e versioni superiori, è possibile lavorare anche mediante le cosiddette *stream*, che aprono alla possibilità di eseguire un calcolo concorrente/parallelo, dunque più performante di un calcolo sequenziale, su un moderno computer multi-core.
- Una stream non è una nuova collezione (con copia degli elementi); piuttosto è una *vista* su una collezione di dati esistente.
- Diversi metodi sono disponibili per elaborare stream, che di norma generano come risultato una nuova stream etc. sino a quando non si applica un'*operazione terminale* (es. di *riduzione*) che finalizza il calcolo ed attiva di fatto le *operazioni intermedie*.

```
ls.stream()
        .filter( s -> s.length()>=3 && s.contains("b") ) //operazione intermedia
        .forEach( System.out::println ); //operazione terminale con ref method
```

Si genera una stream da ls, si filtrano gli elementi con una lambda che realizza un **Predicate** il cui metodo *test(Object)* seleziona quegli elementi che soddisfano una data condizione; si ottiene cosi una nuova stream che quindi viene stampata. Le operazioni intermedie non vengono applicate sino a quando non si incontra l'operazione terminale, in questo caso la forEach.

```
long c=ls.stream()
        .filter( s->s.length()>=3 && s.contains("b") )
        .count(); //una reduce operation come operazione terminale
```

Qui si richiede di ritornare il conteggio degli elementi che soddisfano il predicato. Altre volte si può chiedere di trasformare la collezione da list di stringhe a lista es. di interi:

```
ls.stream()
        .filter( s->s.length()>=3 && s.contains("b") )
        .mapToInt( s -> s.length() ) // qui si genera una stream delle lunghezze
        .forEach( System.out::println );
```

Data una collezione di persone (classe Persona) lp, ciascuna delle quali ha un'eta, sesso, telefono, interessi etc. , più opportuni metodi, ed una persona campione **target,** si può avere:

```
List<String> affini=lp.stream()
        .filter( p->p.getSesso()!=target.getSesso() &&
                Math.abs( p.getEta()-target.getEta() )<=10 &&
                p.interessiInComune(target) )
        .map( p -> p.getTelefono() ) //stream di stringhe nr di telefono
        .collect( Collectors.toList() );
```

Se la collezione lp ha una grande cardinalità, anziché effettuare le operazioni di ricerca/ filtraggio, mapping e raccolta delle informazioni telefoniche in sequenza, si può usare una **parallel stream** che accende più thread per l'effettuazione delle operazioni, in esecuzione parallela su un sistema multi-core. La gestione della concorrenza è qui resa trasparente in quanto è realizzata in modo sicuro dalla gestione delle stream:

```
List<String> affini=lp.stream().parallel() o direttamente: lp.parallelStream()
        operazioni
        …
```

Approfondimenti

- I concetti riguardanti le lambda expression e le stream, possono essere approfonditi, ad es., su
  - Cay Horstmann: *Java 8 for the really impatient*, Addison Wesley, 2014.
  - Angelika Langer & Klaus Kreft: *Lambda expressions in Java – Tutorial*, 2013, http://www.angelikalanger.com/Lambdas/Lambdas.pdf
- Naturalmente, fondamentale resta sempre la consultazione on-line delle API di Java.

Alcune novità introdotte dalle versioni più recenti di Java

- Spesso Java è spesso criticato perché è visto come un linguaggio che «costringe» stili di programmazione «prolissi» che alla fin fine possono «annoiare» il programmatore.
- Nelle versioni più recenti (Oracle ha stabilito una periodicità di 6 mesi per l'uscita di nuove versioni del JDK. Al momento siamo alla versione 15 del linguaggio) diverse estensioni sono state introdotte per «semplificare» lo stile di programmazione e renderlo il più essenziale e compatto possibile, ma al tempo stesso più robusto e sicuro.
- Un esempio è l'istruzione forEach delle nuove versioni delle collezioni che, utilizzando un iteratore interno e una lambda expression, permette di ridurre al minimo il codice da scrivere per applicare una certa operazione (Consumer) su tutti gli elementi della collezione.

La classe **Optional** e la gestione di risultati **null**

- È ben noto che quando un metodo ritorna come risultato un oggetto, il metodo può anche ritornare **null** che se non esplicitamente testato, può dar luogo ad una NullPointerException.
- La classe Optional permette di «non dimenticare» questa possibilità ed offre modi espliciti per trattare sia il caso di restituzione di un risultato non null (caso normale) che quello del null.
- Es. public **Optional<Foo>** method(); anzichè restituire semplicemente Foo.
- A questo punto diventa possible scrivere:
- Foo foo=method()

  .orElse(new Foo());
- In altri termini, se il metodo restituisce un oggetto Foo non null, tutto è ok. Altrimenti (il risultato è null), si può (ad es.) creare un oggetto Foo o altro default.
- È anche possibile esplicitamente sollevare una eccezione in caso di null:
- Foo foo=method()

  .orElseThrow(ClasseEccezione::new);
- La formulazione di method() con tipo del risultato Optional, *obbliga* il programmatore a specificare cosa fare quando il risultato non è presente (il caso null). Naturalmente, il metodo non può più restituire null ma deve creare e restituire un Optional.

Come restituire e interrogare un Optional

- Nel corpo del metodo che ritorna Optional, non si dovrebbe più restituire null ma procedere come segue:
- Optional<Foo> foo=Optional.of(new Foo()); //l'optional è popolato
- Optional<Foo> foo=Optional.empty(); //l'optional è esplicitamente vuoto
- Il chiamante del metodo può quindi sapere se l'oggetto Optional restituito dal metodo è «pieno» o «vuoto» con le query booleane:
- foo.isPresent() o foo.isEmpty() dopo di che se l'Optional è pieno si può ottenere l'oggetto vero risultato con get:
- if(foo.isPresent()){ Foo ris=foo.get(); }

  else{ gestione del caso l'Optional è vuoto }

Type inference sulle variabili locali

- Sempre rimanendo un linguaggio a tipizzazione statica, ossia uno nel quale il tipo delle variabili è noto a tempo di compilazione e non dinamicamente come accade ad es. in Python, nelle ultime versioni il compilatore è in grado di inferire il tipo delle sole **variabili locali** dall'assegnazione di valore, per cui l'utente può snellire il relativo codice. A questo proposito è stata resa disponibile la parola riservata **var** utilizzabile come segue:
- **var lis=new ArrayList<String>();** //lis ha tipo ArrayList<String>
- Ma una dichiarazione **var v;** dà luogo a una segnalazione di errore da parte del compilatore in quanto non è noto il tipo di v. Stesso comportamento sussiste quando una dichiarazione **var** fosse introdotta nella testata di metodi o costruttori per i parametri formali.
- Il meccanismo è utilizzabile esclusivamente all'interno di metodi o di blocchi di istruzioni. Un suo uso efficace si basa necessariamente su una scelta oculata dei nomi delle variabili. Una istruzione tipo:
- **var x=getX();**
- è da evitare perchè sebbene il compilatore riesca ad inferire il tipo di x dal tipo di ritorno di getX(), un lettore avrebbe difficoltà a capire il tipo della variabile.
- I progettisti di Java, tuttavia, suggeriscono di non esagerare con l'uso delle var.

Nuove operazioni sulle stream delle collection

- A partire da Java 9, sulle stream ottenibili dalle collection si possono applicare anche le nuove operazioni: **takeWhile()** e **dropWhile()**. Esse ricevono un Predicate (dunque spesso una lambda) che sin tanto che si mantiene true dà senso all'operazione.
- **takeWhile(condition)** mantiene nella stream gli elementi che soddisfano il predicato; l'operazione si interrompe sul primo elemento che non soddisfa.
- **dropWhile(condition)** fa l'opposto: rimuove dalla stream gli elementi che soddisfano la condition. L'arrivo di un elemento che non verifica la condition interrompe l'operazione.
- Segue un esempio relativo a dei conti correnti bancari. Tutti quelli il cui tipo è CHECKING si vuole tralasciarli nella:

```
var customvar accountsList = getAccounts();
var checkingAccountsList = accountsList .stream()
        .dropWhile( account -> {return account.type()==CHECKING;} )
        .collect( Collectors.toList() ;
```

switch expression

- Già da tempo, l'istruzione switch può basarsi su stringhe anziché solo valori discreti (caratteri, interi, enumerazioni etc.):

```
switch(nome){
    case "cane": …; break;
    case "casa": …; break;
}
```

- A partire da Java 12, il linguaggio consente di vedere uno switch come una espressione che produce un valore. Lo switch che segue assegna a id un codice che dipende dal valore della stringa name (si noti l'uso del simbolo di mapping -> come nelle lambda):

```
String id = switch(name) {
    case "john" -> "12212";
    case "mary" -> "4847474";
    case "tom" -> "293743";
    default -> "";
};
```

- Si nota esplicitamente che non occorre usare **break** per terminare un'alternativa. Il significato di restituire un valore (->), infatti, elimina l'effetto a cascata di un normale switch, impedito appunto da **break**.

- E’ comunque possibile combinare alternative come illustrato di seguito:

```
return switch(name) {
        case "john", "demo" -> "12212";
        case "mary" -> "4847474";
        case "tom" -> "293743";
        default -> "";
};
```

- Il corpo di una alternativa (dopo ->) può consistere di una coppia { e } per specificare in modo più generale il valore da ritornare.
- Si nota, infine, che una o più alternative possono anche sollevare eccezioni:

```
return switch(name){
        case "john" -> "12212";
        case "mary" -> "4847474";
        case "tom" -> "293743";
       default -> throw new InvalidNameException();
};
```

Record

- I record rappresentano un nuovo tipo di classe introdotto in Java 14. I record sono ideali per rappresentare classi che consistono solo di campi e di accessi a tali campi.
- Un esempio di record è il seguente:

public record Conto(int id, int customerId, String tipo, double bilancio) {}

- Si nota l'uso della parola chiave **record** invece di **class**. I campi sono definiti all'interno di una coppia di (e), separate da virgole.
- La "burocrazia" termina qui. Non occorre specificare i metodi getter/setter nè ridefinire i metodi canonici toString(), equals() e hashCode() da momento che essi sono già provvisti di default.
- Se si intende ridefinire, ad es., il metodo toString() o aggiungere nuovi metodi, ciò può essere fatto all'interno del corpo **{}.**

```
public record Account( int id, int customerId, String type, double balance ) {
    @Override
    public String toString(){ return “ridefinizione di toString()"; }
}//Account
```

L’instanziazione di un record si fa esattamente come per le classi:

```
Account acc=new Account( 1256, 11554, ”ORDINARIO”, 5000.0 );
```

I record sono **immutabili**. Ossia, i campi sono implicitamente final.

Per accedere ad un campo, la notazione disponibile usa direttamente il **nome** del campo, ad esempio:

```
String c=acc.tipo();
```

e non acc.getTipo() che è usuale con le classi.

Essendo final, i record **non possono ereditare** da altre classi o record, ma possono eventualmente implementare una o più interface.

La notazione è chiarita di seguito. Data l'interfaccia:

```
public interface AccountInterface { void metodo(); }
```

si può avere:

```
public record Account( int id, int customerId, String tipo, double bilancio )
    implements AccountsIn{
        public void metodo(){…}
}
```

Blocchi di testo

A partire dalla versione 13, Java consente di scrivere blocchi di testo complessi, evitando gli usuali problemi di una lunga stringa che ad es. non può continuare tra più linee. Piuttosto, si tratta sempre di esprimere frammenti di linea chiusi tra " e " e concatenati con "+", includendo le sequenze di escape ‘\n‘ al fine di programmare nella stringa complessiva tutte le andate a capo.
Attualmente, un stringa complessa (*blocco di testo*) può essere espressa in modo più naturale all’intero di """ e """, (tre doppi quotation mark) senza le complicazioni delle sequenze di escape per comandare l’andata a capo. All’interno di un blocco di testo i fine linea e i " sono rispettati per come l’utente digita il testo.
Schema di un text block:

```
"""
line1
line2
line3
"""
```

o al più (sebbene poco usata come struttura):

```
"""
line1
line2
line3 """
```

```
String txt=
"""
Un esempio
     di testo
          indentato
""";
```

è equivalente allo schema più tradizionale:

```
String txt=
     "Un esempio\n" +
     "      di testo\n" +
     "           indentato\n";
```

Si potrebbero usare gli escape tabulatori

```
String text = """
Inizio testo \
che continua anche suddiviso \
su piu' linee.\ """;
```

Qui si evitano i fine linea

```
String text = "Inizio testo " +
"che continua anche suddiviso " +
"su piu' linee.";
```

Versione tradizionale

```
String html = """
<HTML>
  <BODY>
    <H1>"Java 13 in action!"</H1>
  </BODY>
</HTML>""";
```

corrisponde all'usuale testo:

```
String html =
"<HTML>\n\t<BODY>\n\t\t<H1>\"Java 13 in action!\"</H1> \n\t</BODY>\n</HTML>\n";
```

\n è l'escape fine linea
\t è l'escape tabulatore
\" è l'escape per includere il " nella stringa testo.