INGEGNERIA DEL SOFTWARE

Lez.1

AGGIORNAMENTI DI JAVA NEL TEMPO

Da Java 8, il linguaggio permette il supporto alla **programmazione funzionale**, essa permette di ridurre notevolmente le righe di codice necessarie senza rinunciare alla **chiarezza** delle operazioni che vogliamo eseguire. Uno dei principali vantaggi della programmazione funzionale, oltre all'espressività. è il supporto alla **programmazione parallela,** ovvero, essa ci permette di sfruttare sistemi multicore in modo più semplice rispetto alla programmazione ad oggetti: in un futuro dove la potenza di calcolo ci è data dall'aggiunta di core, questo paradigma diventa essenziale per sfruttare l'hardware, è inoltre importante ricordare che meno righe di codice sono presenti, meno test dobbiamo effettuare.

Un qualsiasi linguaggio, per continuare ad essere usato nel tempo, deve offrire **supporto** agli utenti e soprattutto aggiornarsi in base alle esigenze dei programmatori, aggiungendo delle specifiche che rendano più semplice la programmazione.
Molti di questi aggiornamenti e migliorie, vengono forniti tramite il cosiddetto **zucchero sintattico**, ovvero, vengono introdotte nuove **sintassi**, semplificate e più sicure da usare per operazioni già possibili ma con altre notazioni, la caratteristica di queste aggiunte sintattiche è che non aggiungono funzionalità ma semplificano quelle già esistenti. Sono esempio di zucchero sintattico i **cicli for enanched** e la possibile omissione del tipo quando si dichiara una list (il tipo viene dedotto dal compilatore tramite **type inference,** ovvero, il compilatore indovina il tipo in base al contesto).

ESPRESSIONI LAMBDA

Un'espressione lambda è una **funzione anonima** che prende in ingresso parametri con nome dato a sinistra del segno della **freccia** e un blocco di codice a destra del segno della freccia. **La particolarità delle espressioni lambda è che possono essere passate come argomento di un metodo** sia passando direttamente l'espressione che memorizzando il return dell'espressione in una variabile.

Nei linguaggi funzionali puri, un'espressione lambda, è una **funzione pura,** ovvero, il risultato dipende soltanto dagli input (non ha uno stato e il risultato non può essere influenzato da degli stati esterni). In Java, un'espressione lambda può accedere a uno stato esterno, dunque può anche essere non pura.
La seguente è un'espressione lambda che prende in ingresso due parametri, x e y, e implementa la somma, il risultato è il valore di ritorno.

(x, y) - > x + y

I parametri in ingresso sono x e y (**il tipo è identificato automaticamente**), quando la funzione non ha parametri, o ne ha più di uno, occorre racchiudere i parametri tra parentesi tonde.

( ) - > System.out.println(“Hello World!”)

Il corpo della funzione anonima è il codice a destra della freccia, nel caso di più di un’istruzione, occorre racchiuderle tra **parentesi graffe** e specificare quale istruzione è quella da ritornare.

(x, y) - > { System.out.println(“x: “+x); return x+y }

UTILIZZO DELLE ESPRESSIONI LAMBDA

Un utilizzo molto diffuso del paradigma funzionale avviene nell’ambito delle **interfacce Java** con un solo metodo (**interfacce funzionali**), infatti, anziché definire tramite **override** il metodo in una classe, posso usare un’espressione lambda che **definisce il comportamento di quel metodo**, senza dover implementare alcuna classe specifica, in questo caso il tipo di ritorno è definito dall’interfaccia e non viene dedotto a compile-time. Si ricorda che non è possibile usare le espressioni lambda per definire il comportamento di un’interfaccia che presenta più di un metodo (non funzionali), infatti, il compilatore non sarebbe in grado di capire a quale metodo si stia riferendo l’espressione.

PROGRAMMAZIONE DICHIARATIVA VS PROGRAMMAZIONE IMPERATIVA

In genere la programmazione funzionale rientra all’interno della **programmazione dichiarativa**. In genere Java è un linguaggio imperativo anche se esistono **API** e **librerie** che ci permettono di programmare in modo dichiarativo.

ESEMPIO:

```
public class Trova {
	private List<String> listaNomi = Arrays.asList("Nobita", "Nobi","Suneo", "Honekawa",
"Shizuka", "Minamoto", "Takeshi", "Gouda");

// in stile imperativo
public void trovaImper() {
	boolean trovato = false;
	for (String nome : listaNomi)
		if (nome.equals("Nobi")) {
			trovato = true;
			break;
		}
	if (trovato) System.out.println("Nobi trovato");
	else System.out.println("Nobi non trovato");
}
```

```
// in stile dichiarativo
public void trovaDichiar() {
        if (listaNomi.contains("Nobi")) System.out.println("Nobi trovato");
        else System.out.println("Nobi non trovato");}
}
```

Lo stile imperativo dà al programmatore il controllo di quel che il programma deve fare, tuttavia:

•Bisogna implementare varie linee di codice, e molto spesso si hanno cicli che scorrono liste per trovare valori, calcolare somme, etc.

•Per capire cosa si vuol fare, dobbiamo prima leggere tanti dettagli all'interno del corpo del ciclo.

•Il ciclo è esterno al codice che implementa la lista (l'iterazione è esterna)
Nella versione dichiarativa vari dettagli dell'implementazione sono nella libreria sottostante (sul metodo contains() della classe ArrayList l'iterazione è interna. Data la natura di Java, all'interno della libreria, l'implementazione sarà sicuramente in stile imperativo, semplicemente è trasparente (e quindi più semplice) da usare.

STILE FUNZIONALE

•Lo stile di programmazione funzionale è dichiarativo. La programmazione funzionale aggiunge allo stile dichiarativo funzioni di ordine più alto, ovvero funzioni che hanno come parametri altre funzioni.

•In Java si possono passare oggetti ai metodi, creare oggetti dentro i metodi, e ritornare oggetti dai metodi.

•In Java 8 si possono passare funzioni ai metodi, creare funzioni dentro i metodi e ritornare funzioni dai metodi.

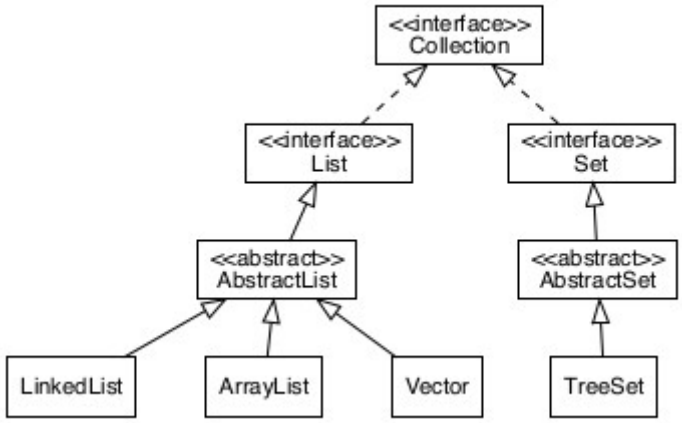
•Un metodo è una parte di una classe, mentre una funzione non è associata ad una classe.

•Un metodo o una funzione che ricevono, creano o ritornano una funzione, si considerano essere funzioni di ordine più alto

Lez. 2

JAVA COLLECTION

Un altro uso della programmazione funzionale in Java si riscontra nella manipolazione delle **strutture dati** implementate all'interno delle **librerie** del linguaggio.
Tutte le strutture sono organizzate in una **gerarchia**. In cima a questa gerarchia si trova l'**interfaccia Collection** che definisce dei **metodi astratti** (senza implementazione) che ricoprono le **operazioni tipiche** che si possono fare su una generica collezione di oggetti (es: add(), remove(), get(), set()). Essendo in questa interfaccia, tutti questi metodi saranno presenti (ereditati) e implementati in tutte le **sottoclassi** più in basso nella gerarchia.



Quando utilizziamo le librerie di Java, non abbiamo bisogno di preoccuparci dell'implementazione delle strutture, dobbiamo però porre molta attenzione alle complessità delle strutture che stiamo utilizzando, scegliendo sempre quella più adatta al task da compiere, in questo modo, al programmatore, non viene dato l'onere di implementare e testare una struttura ogni volta che ne ha di bisogno.

Esempi di strutture già presenti:

• **ArrayList** è un array espandibile: cresce del 50% quando non vi è spazio, gli elementi sono contigui, quindi l'accesso al generico elemento è veloce.

• **LinkedList** è una lista, ogni elemento ha i riferimenti al successivo e al precedente.

• **Vector** è simile ad ArrayList , ma è **synchronized,** quindi **thread safe,** ovvero, automaticamente gestisce l'accesso concorrente evitando così **race condition.**

METODI DI DEFAULT

Java 8 introduce i metodi di default per le interfacce, ciò ha permesso di includere nuovi metodi su interfacce già esistenti, senza compromettere la compatibilità delle applicazioni conformi a precedenti versioni di Java. **stream()** è un **metodo di default** dell'interfaccia **Collection,** dunque, sarà implementato da tutti i suoi sottotipi. Questo comportamento è inusuale, infatti, abbiamo aggiunto del codice all'interno di un'interfaccia, questa "violazione" è stata fatta per evitare che l'utilizzatore debba riscrivere il metodo. La particolarità di questi metodi è che **essi non possono intervenire sullo stato** a differenza dei metodi implementati sulle **classi astratte** che invece possono farlo.

Un particolare e comune metodo di default è come detto prima **stream()**, esso ci permette di interagire tramite la programmazione funzionale con le strutture dati implementate a partire da Collection.

stream() restituisce uno **Stream<T>** che è una **sequenza di elementi di tipo T** (T è definito dalla collection su cui è invocato stream()). Stream() permette di fare operazioni sugli elementi presenti sull'istanza di collection su cui è invocato.

Esempio: chiamato su una lista, il metodo stream restituisce una sequenza contenente gli elementi della lista e il tipo di ritorno sarà uno stream.

Il tipo Stream definisce vari metodi che prendono in ingresso funzioni lambda.

Uno dei metodi messi a disposizione da stream è **filter().**

METODO FILTER

Possiamo immaginare lo Stream come un flusso di elementi e **filter** un filter

Si abbia una lista di String , contare quante volte è presente un certo valore.

```
List<String> nomi = List.of("Nobita", "Nobi", "Suneo");
long c = nomi.stream()
                        .filter(s -> s.equals("Nobi"))
                        .count();
```

Il metodo filter viene definito all'interno del tipo stream e ci permette di applicare un criterio di filtro per gli elementi presenti in uno stream, esso non fa altro che imporre una condizione che, presi gli elementi di uno stream, pone in uno stream di uscita tutti gli elementi dello stream in input che rispettano la condizione dettata da **filter** stesso.

filter(Predicate<T> p) è un metodo di Stream , prende come argomento una funzione che ritorna un **boolean** (quindi solo true o false). **Una funzione che ritorna un boolean è detta predicato**.
Filter() ritorna uno Stream costituito da tutti gli elementi della lista che soddisfano il predicato passato in input.

L'**espressione lambda** s -> s.equals("Nobi") è un **predicato** che ha in ingresso s , che è un

elemento dello stream, e restituisce un boolean. Inoltre, equals() è un metodo di String che restituisce un boolean.

•filter() è **lazy**, (oppure **operazione intermedia**), ovvero, essa può essere eseguita dal compilatore soltanto quando necessario e non necessariamente nel flusso stabilito dal codice.

•**count()** è un metodo di Stream **eager** (oppure, **operazione terminale**), essa genera un valore e forza l'esecuzione delle precedenti operazioni.
Tutte le operazioni che restituiscono uno Stream sono operazioni intermedie mentre quelle che non restituiscono stream sono dette terminali.

ESEMPI CON FILTER

Contare quanti elementi della lista nomi hanno lunghezza 5 caratteri:

long c = nomi.stream().filter(s -> s.length() == 5).count();

L'espressione lambda **s -> s.length() == 5** ha in ingresso s , ovvero un elemento dello stream, su s si invoca l**ength()** (metodo di String che restituisce la lunghezza di s), quindi si valuta se è pari a 5.

TIPO PREDICATE

Si può definire una funzione che restituisce un boolean:

Predicate<Integer> positive = x -> x >= 0;

Predicate rappresenta un'**interfaccia funzionale**, ovvero un'interfaccia che definisce un solo metodo.
L'annotazione @**FunctionalInterface** indica al compilatore che l'interfaccia ha solo un metodo astratto, così il **compilatore** controlla che abbia un solo metodo (evita che versioni successive inseriscano altri metodi).

Predicate definisce il metodo **test()** che prende in ingresso un parametro di tipo Object, iin questo esempio, il metodo test() ha parametro in ingresso x di tipo Integer e la sua implementazione è x >= 0;

Il predicato positive si può passare ad un metodo:

Stream<Integer> result = Stream.of(2, 5, 10, -1).filter(positive);

Esempio

```
// Conta tutti gli elementi non vuoti (lunghezza > zero)
   public long es0(List<String> list) {
       return list.stream().filter(s -> !s.isEmpty()).count();
   }
```

Lez. 3

TIPO PREDICATE

Predicate è un'interfaccia funzionale, ovvero, un'interfaccia all'interno della quale è definito un solo metodo che prende il nome di **test()**. Il tipo predicate viene passato in input a stream. Ricordiamo che a livello sintattico l'implementazione di questo metodo viene fornita tramite un'espressione lambda.

Esempio:

Predicate<Integer> positive = x -> x > 0;

In questo esempio, il metodo test() ha parametro in ingresso x di tipo Integer e la sua implementazione è **x > 0.** Il predicato positive, si può passare ad un metodo:

Stream<Integer> result = Stream.of(2, 5, 10, -1)
.filter(positive);

.of() è un metodo statico della classe stream che costruisce (restituisce uno stream) con i metodi passati. Notiamo che stream.of è diverso da List.of: la differenza sta nel fatto che uno stream, una volta che viene attraversato (utilizzato) viene chiuso e non può più essere utilizzato.
Se provassimo a riutilizzare uno stream, il risultato sarebbe un'eccezione a runtime "I**llegal state exception**" poiché abbiamo già modificato lo stato dello stream visitandolo la prima volta. Questo errore vale anche per i metodi non terminali.
La situazione è diversa nel caso di operazioni concatenate: in questo caso stiamo manipolando una sola volta lo stream ma in **cascata.**
Si potrebbe pensare di lavorare su delle **copie** dello stream ma, da libreria, non è definito un **copyOf()**, non è dunque possibile creare copie del tipo stream. Infatti, ogni volta che utilizziamo un'operazione non terminale, lo stream restituito è uno **stream differente**.

METODO REDUCE

reduce(T identity, BinaryOperator<T> accumulator) è un metodo di Stream , prende un **valore** dello **stesso tipo** degli elementi dello stream, e **un'espressione lambda** (binaria) che ha due valori in ingresso e ritorna un valore. Il metodo reduce() viene utilizzata quando vogliamo passare da un insieme di valori ad un singolo valore, ad esempio per ottenere la somma di un insieme. (**count()** è un particolare tipo di reduce()).

Esempio:

reduce(0, (accum, v) -> accum + v);

In questo esempio, **0** è il **valore di partenza** e l'espressione lambda è la funzione che effettua la somma, userà dunque **accum** per memorizzare la somma mentre ciascun valore dello stream è rappresentato da **v**.Nel caso in cui lo stream fosse vuoto, il risultato sarebbe il valore passato, ovvero 0.
reduce() è un'**operazione terminale.**
Esempio di codice:

```
public class Pagamenti{
      private List<Float> importi = new ArrayList<>();

      //in stile imperativo
      public float calcolaSommaImper(){
          float risultato = 0f;
          for(float v : importi)
              risultato +=v;
          return risultato;
      }

      //in stile funzionale
      public float calcolaSomma(){
          return importi.stream()
                        .reduce(0f, Float::sum);
      }

  }
```

TIPI WRAPPER

I tipi wrapper sono degli oggetti al cui interno possiamo inserire tipi primitivi, permettono dunque di chiamare metodi su dei tipi primitivi usando i rispettivi tipi wrapper.

Esempio:

```
      public static void Wrapper(){
          Integer a = 5; //BOXING (wrapper <- tipo primitivo)
          int b = a;     //UNBOXING (primitivo <- wrapper)
      }
```

Qui notiamo che grazie a dello zucchero sintattico, è possibile assegnare un tipo primitivo ad un oggetto tramite l'operazione **=**, ovviamente, si sottintende l'instanziamento di un oggetto tramite la parola chiave new. (Integer a = new Integer(5);).

L'operazione di inserire un tipo primitivo all'interno di un wrapper viene detto **boxing,** ovviamente, questa sintassi, nasconde delle operazioni nascoste.
L'operazione di assegnare un wrapper all'interno di un tipo primitivo viene detta invece **unboxing.**

RIFERIMENTI A METODI

Dall'esempio precedente notiamo **Float::sum,** osserviamo un paio di concetti:

La sintassi **::** permette di riferirsi ad un metodo static della classe che si può chiamare da un'altra classe, in questo caso, sum è un metodo presente all'interno della classe wrapper Float.

Esiste un **overload** di reduce che non prende in input il valore iniziale (0 nel nostro caso):

```
Optional<Float> var = list.stream()
                      .reduce(Float::sum);
```

Questo metodo, anzichè generare **un'eccezione**, ritorna un tipo **Optional<Class>** ovvero un Wrapper che potrebbe contenere valori NULL. (Optional<Float> nel nostro caso). Infatti, non passando in input nessun valore iniziale, lo stream su cui chiamiamo sum potrebbe essere vuoto e quindi saremmo impossibilitati nel fare la somma. Essendo Optional un tipo delicato, su di lui è definito il metodo **isPresent()** che restituisce un **booleano** che indica la presenza o meno di dati all'interno della variabile, dopodichè, per estrarre il valore contenuto al suo interno si utilizza il metodo **get().** Tutti questi metodi sono utilizzati per assicurarsi che non si generino errori manipolando dati che potrebbero lanciare **NullPointerException,** questo tipo di programmazione è detta **Programmazione difensiva**, ovvero, obbliga il compilatore ad effettuare dei controlli.

Inoltre, è importante ricordare che possiamo definire un metodo static invocabile tramite :: in qualunque classe, basta specificare la sintassi **nomeClasse::nomeMetodo;**
Definire questa tipologia di metodi è utile nel momento in cui l'operazione da effettuare all'interno della reduce è molto complessa, quindi non esprimibile tramite espressioni lambda oppure non è definita all'interno di nessuna libreria.

In sintesi esistono tre modi di passare la funzione a **reduce(),** ovvero:

- -Tramite **espressione lambda**;
- -Tramite **metodo di libreria** usando l'operatore **::** ;
- -Tramite un **metodo definito da noi** e l'utilizzo di **::** ;

METODO MAP

Il metodo map è un altro metodo del tipo stream che prende in input una funzione, ovvero, un tipo Function. Function è un'interfaccia funzionale che prende un tipo compatibile con il tipo dello stream e ritorna un altro tipo.
Il metodo map permette appunto di mappare ogni elemento dello stream in un altro tipo ricavato dalla funzione.

In altre parole, **map(Function<T, R> mapper)** di Stream prende in ingresso una **funzione mapper**, e restituisce uno stream contenente i risultati dell'esecuzione della funzione su ciascun elemento dello stream iniziale, ovvero, map() chiama su ciascun elemento dello stream iniziale la funzione passata e dà in uscita il risultato della funzione. Ciascun risultato è inserito in un nuovo stream, **Ad esempio, potrei avere uno stream di persone e potrei ottenere in output, tramite map, uno stream di età delle persone.**
E' importante ricordare che map è un'**operazione intermedia**, restituisce dunque uno stream

Esempio:

map(Persona::getEta);

Eseguito su uno stream di istanze di Persona , restituisce uno stream contenente i valori in uscita da getEta() , quest'ultimo è un metodo di Persona ed è invocato su ciascun elemento dello stream iniziale che data una persona ne restituisce l'età.

Codifica:

```
public class Persona {
	private String nome;
	private int eta;
	public Persona(String n, int e) {
		nome = n;
		eta = e;
	}
	public String getNome() {
		return nome;
	}
	public int getEta() {
		return eta;
	}
}

List<Persona> p = Arrays.asList(new Persona("Saro", 24),
new Persona("Taro", 21), new Persona("Ian", 19),
new Persona("Al", 16));
```

```
//Calcoliamo la somma delle età in stile funzionale
int somma = p.stream()
       .map(Persona::getEta)
       .reduce(0, Integer::sum);

//Stile imperativo
int somma = 0;
for (Persona x : p)
   somma += x.getEta();
```

Questo stile è alla base del **modello map-reduce**, un modello molto usato nell'analisi di grosse quantità di dati in parallelo, questo perchè, passando a reduce **un'operazione associativa** (l'ordine degli operandi non importa), possiamo **parallelizzare** le operazioni rendendo molto più veloce il calcolo su **un'architettura parallela**. Accenniamo al fatto che gli stream paralleli non fanno uso di thread, sono quindi due tipologie di parallelismo differenti.

Un grosso vantaggio di questo stile è la totale eliminazione di **cicli for** che potrebbero rendere tedioso il codice.

ESEMPIO: RICERCA SU UNA LISTA

Data la lista di istanze di Persona , trovare il nome della persona che è più grande (di età) fra quelli che hanno meno di 20 anni

```
List<Persona> p = Arrays.asList(new Persona("Saro", 24),
new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al",16));
```

Ricerca imperativa

```
Persona pmax = null;
for(Persona x : p){
   if(x.getEta() < 20){
       if(pmax == null )    pmax = x;
       if(pmax.getEta() < x.getEta())   pmax = x;
   }
   if(pmax != null)     System.out.println("persona" + pmax.getNome());
}
```

Il corpo del ciclo ha varie condizioni, queste rendono il codice più difficile da comprendere.
Ricerca funzionale versione 1

```java
/** restituisce l'istanza con il valore massimo di eta' */
public static Persona getMax(Persona p1, Persona p2) {
   if (p1.getEta() > p2.getEta())
   return p1;
   return p2;
   }
Optional<Persona> pmax = p.stream()
                         .filter(x -> x.getEta() < 20)
                         .reduce(Persona::getMax);

if(pmax.isPresent())
   System.out.println("persona: " + pmax.get().getNome());
```

-filter() è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento;

-reduce() è usata per selezionare un elemento: invoca getMax() che confronta a due a due.

Ricerca funzionale versione 2

```java
Optional<Persona> pmax = p.stream()
                         .filter(x -> x.getEta() < 20)
                         .max(Comparator.comparing(Persona::getEta));

if(pmax.isPresent())
   System.out.println("persona: " + pmax.get().getNome());
```

-filter() è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento;

-max() è un metodo del tipo stream che trova il valore massimo, è un'operazione terminale, prende un **Comparator**, restituisce un Optional , **max()** opera in modo simile a **reduce();**

-Un **comparatore** è una funzione che serve a controllare **l'ordinamento di due oggetti:** restituisce **-1** se il primo elemento è maggiore del secondo, **0** se i due elementi sono uguali e **+1** se il secondo elemento è maggiore del secondo, implementa dunque una relazione di ordinamento.

-comparing() è un metodo di Comparator che estrae una **chiave** data in input una funzione che darà il metro di paragone, in questo esempio, dato in ingresso lo stream delle persone, compara l'età delle persone e grazie a max() restituisce il massimo.

METODO COLLECT

Il metodo **collect** è un metodo di stream che ci permette di prendere gli elementi di uno stream e **raggrupparli** (collezionarli) all'interno di una **collection**, ad esempio permette di prendere uno stream e restituire una lista contenente una lista contenente tutti gli elementi dello stream. E' molto utile quando, dopo aver manipolato lo stream, vogliamo ottenere una struttura dati su cui poter lavorare.

Esempio:

```
List<Persona> p = List.of(new Persona("Saro", 24),
new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al",16));

List<Integer> e = p.stream()
                   .map(x -> x.getEta())
                   .collect(Collectors.toList());
```

-**collect()** permette di raggruppare i risultati e prende in ingresso un Collector;

-La classe **Collectors** implementa metodi utili per raggruppamenti, il metodo **toList()** restituisce un Collector che accumula elementi in una List

-map() restituisce uno stream con i valori delle età, e la funzione passata dice come trasformare ciascun elemento dello stream.

PROGRAMMAZIONE PARALLELA

I processori attuali hanno vari **core**, tipicamente due per i portatili, quattro per i desktop, dodici per i server. La programmazione parallela è più difficile di quella sequenziale e usare bene l'hardware risulta più complicato. In Java, la classe **Thread** permette di lanciare un nuovo thread di esecuzione, ma spesso bisogna risolvere i problemi di **corsa critica**, usando opportunamente **synchronized**, **wait**, **notify**.

Le operazioni **map()** e **filter()** di Stream sono **stateless**, ovvero **non tengono uno stato durante l'esecuzione**, quindi il risultato non dipende dall'ordine in cui i singoli elementi su cui eseguono vengono prelevati. Lo stream di origine non viene modificato e le operazioni map() , filter() , etc., generano uno stream distinto. **Questo facilita grandemente la parallelizzazione**

Attenzione: se l'applicazione durante l'esecuzione di un'operazione, per es. di map() , aggiorna uno **stato globale,** l'operazione non è più stateless, inoltre non possiamo effettuare operazioni **non associative** in parallelo perché otterremmo risultati non corretti.

STREAM PARALLELI

Collection ha i metodi di default **stream()** e **parallelStream().**

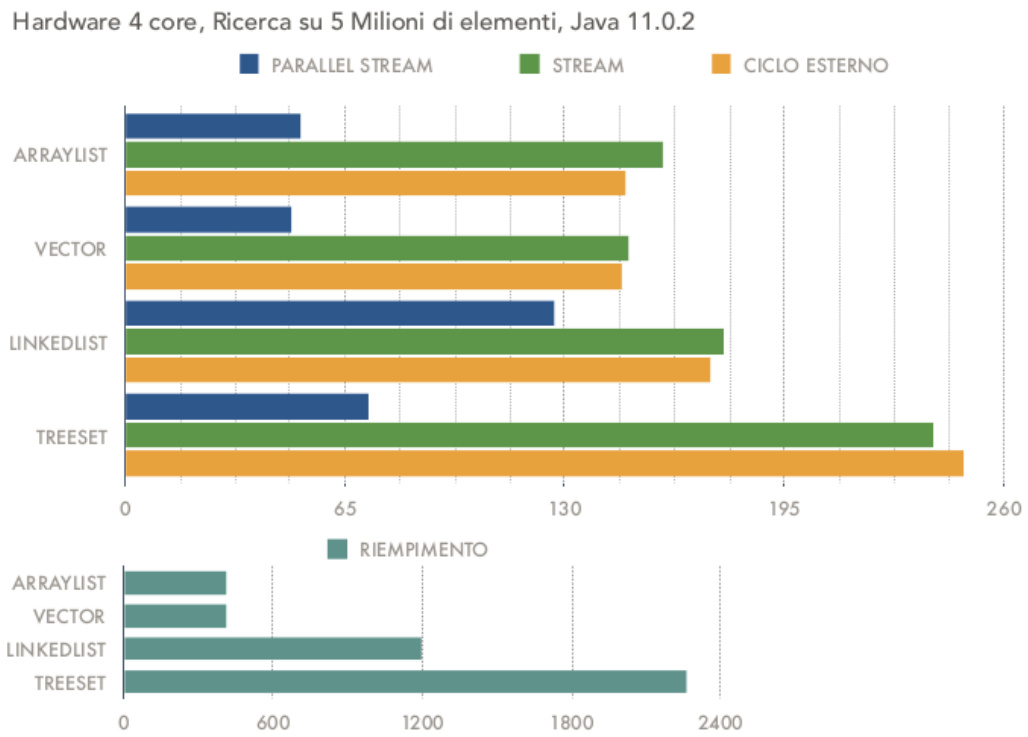
Il metodo **parallelStream**() **possibilmente** dà uno stream parallelo. Quindi si può avere l'esecuzione parallela basata su parallelStream() (niente più bisogno di thread e sincronizzazione, per molti casi)

Le **prestazioni** dipendono da:

- numero di elementi dello stream;
-operazioni da svolgere;
-hardware;
-tipo di Collection su cui si invoca l'operazione.

Eseguendo su milioni di elementi il seguente codice, le prestazioni migliorano (su hardware con più core) per ArrayList, Vector, TreeSet; migliorano poco (o non migliorano) quando si usa **LinkedList** a causa dell'accesso sequenziale.

```
long c = nomi.parallelStream()
            .map(String::toUpperCase)
            .filter(s -> s.equals("NOBI"))
            .count();
```



Nel seguente esempio, noteremo che, eseguire un'operazione non associativa in modo parallelo porterà a dei risultati errati e non prevedibili:

```
    public static void paraAssocReduce(){
        //reduce con operazione associativa (somma)
        List<Integer> list = List.of(10,8,2);
        int res1 = list.parallelStream()
                       .reduce(0, (acc, x) -> acc + x);
        System.out.println(res1);    //stream: 20, parallel: 20

        //reduce con operazione non associativa (sottrazione)
        int res2 = list.parallelStream()
                       .reduce(0, (acc, x) -> acc - x);
        System.out.println(res1);    //stream: -20, parallel: -4
    }
}
```

Lez. 4

ESEMPIO DI RIEPILOGO (forEach e sorted)

Data una lista di istanze di Persona trovare i nomi delle persone che sono giovani ed hanno ruolo Programmer, e ordinare i risultati.

```
List<Persona> team = List.of(new Persona("Kent", 29, "CTO"), new
Persona("Luigi", 25,
                "Programmer"), new Persona("Andrea", 26, "GrLeader"),
new Persona("Sofia", 26,
                "Programmer"));


           team.stream()
           .filter(p -> p.giovane())
           .filter(p -> p.isRuolo("Programmer"))
           .sorted(Comparator.comparing(Persona::getNome))
           .forEach(p -> System.out.print(p.getNome() + " "));

           // Output: Luigi Sofia
```

-**filter():** è un'operazione intermedia che restituisce gli elementi che soddisfano il predicato passato.

-**sorted():** operazione intermedia stateful che restituisce uno stream che ha gli elementi ordinati in base al Comparator passato. QUesta operazione è stateful, ovvero, agisce su tutti gli elementi dello stream, questa operazione potrebbe dare problemi in operazioni parallele perchè prima di essere eseguita tutti i passi precedenti devono essere completati.

-**comparing()**: permette di estrarre la chiave per il confronto.

-forEach(): operazione terminale che esegue un'azione su ciascun elemento dello stream (su uno stream parallelo l'ordine 1 non è garantito).

ALTRO ESEMPIO DI RIEPILOGO (distinct e findAny)

Data una lista di istanze di Persona trovare i diversi ruoli

```
team.stream()
   .map(p -> p.getRuolo())
   .distinct()
   .forEach(s -> System.out.print(s+" "));
```

-distinct(): operazione intermedia stateful che restituisce uno stream di elementi distint.

Data una lista di istanze di Persona trovare il nome di una che ha il ruolo Programmer

```
Optional<Persona> r = team.stream()
           .filter(p -> p.isRuolo("Programmer"))
           .findAny();
   if (r.isPresent()) System.out.println(r.get().getNome());
```

-findAny(): (simile a findFirst()) operazione terminale che restituisce un Optional , per valutarla non è necessario esaminare tutto lo stream, si dice **short-stream** (su uno stream parallelo l'ordine 1 non è garantito) **circuiting** (può far sì che alcune parti non 2 eseguano). Possiamo dire inoltre che questa operazione è **short circuiting** ovvero, potrebbe bloccarsi prima di scorrere tutto lo stream, appena trovato l'elemento che soddisfa la richiesta. Se volessimo trovare il primo elemento, usando però stream paralleli, il metodo da usare sarebbe **findFirst()**, questo perchè dei thread potrebbero arrivare prima di altri. Se la findAny non torna niente, possiamo aggiungere il metodo orElse("string") che torna un messaggio in caso di fallimento.

STATELESS - STATEFUL

•Le operazioni **map()** e **filter()** sono **stateless**, ovvero non hanno uno stato interno, prendono un elemento dello stream e danno zero o un risultato
• Le operazioni come **reduce()**, **max()** accumulano un risultato. Quest'ultimo ha una dimensione limitata, indipendente da quanti elementi vi sono nello stream. Il risultato in una passata viene dato in ingresso alla passata successiva
•Le operazioni come sorted() e distinct() devono conoscere gli altri elementi dello stream per poter eseguire, si dicono **stateful.**

GENERARE STREAM: Iterate()

• Gli stream possono essere generati a partire da una funzione tramite le operazioni **iterate()** e **generate()** . Queste operazioni creano **stream infiniti**.

• **iterate()**: restituisce un stream infinito e ordinato dato dall'esecuzione iterativa di un funzione f applicata inizialmente ad un **elemento seme** , quindi produce uno stream di seme, f(seme) , f(f(seme)) , etc.

• l**imit()**: tronca lo stream ad una lunghezza pari al numero indicato, genera un altro stream della lunghezza indicata.

```
Stream.iterate(2, n -> n * 2)
   .limit(10)
   .forEach(System.out::println);
```

• Il codice sopra dà lo stream di 10 elementi che consiste in 2, 4, 8, ... 1024, ovvero, la potenza del due.

• Ad ogni passo, dopo il primo, il valore in input alla funzione è il valore calcolato dalla funzione al passo precedente.

• L'operazione iterate() è **sequenziale.**

La potenzialità di iterate() è quella di generare stream senza partire da una collection, usando un seme e una funzione. Lo stream generato avrà come primo elemento il seme, il secondo elemento sarà la funzione applicata al seme e tutti gli altri elementi saranno prodotti dalla funzione applicata al risultato precedente.

GENERARE STREAM: generate()

Il metodo **generate()** permette di produrre uno stream infinito di valori, tramite una funzione di tipo **Supplier** , ovvero che fornisce un valore, generate() non applica una funzione ad ogni nuovo valore prodotto, come invece fa iterate().
un **Supplier** è **un'interfaccia funzionale** definita tramite un'operazione lambda che non prende valori in input e restituisce un numero.

```
Stream.generate(() -> Math.round(Math.random()*10))
       .limit(5)
       .forEach(System.out::println);
```

Il codice sopra genera uno stream di 5 **numeri casuali**, ciascuno fra 0 e 10.

TIPO INTSTREAM

IntStream rappresenta uno stream di valori int, è importante ricordare che intStream è un tipo a parte e differente da uno stream di Integer.

```
IntStream.rangeClosed(1, 6)
            .map(x -> x*x)
            .forEach(System.out::println);
```

• Il codice sopra genera e stampa i quadrati dei numeri da 1 a 6.

• **rangeClosed():** restituisce una sequenza di int nell'intervallo specificato (estremi inclusi) con incremento pari a 1.

•**range():** ha lo stesso comportamento di rangeClosed() ma non include l'**estremo destro.**

```
int v = IntStream.rangeClosed(1, 5).sum();
   // Output: v = 15
```

• **sum():** somma i valori presenti nello stream IntStream.

METODI DI INTSTREAM

```
int result = Stream.of("truth", "flows", "to", "them", "sweetly", "by",
"nature")
            .mapToInt(x -> x.length()).sum();
```

-**mapToInt()**: esegue la funzione passata e restituisce un IntStream.

```
  Stream<Integer> s = IntStream.rangeClosed(1, 10).boxed();
```

-**boxed():** restituisce uno Stream di Integer a partire da IntStream.

Esempio:

Si abbia la lista che contiene istanze di Persona, si generi uno stream contenente i primi 4 elementi

```
List<Persona> lista;
   Stream<Persona> p =
   IntStream.rangeClosed(0, 3)
            .mapToObj(i -> lista.get(i));
```

-mapToObj(): restituisce uno stream di oggetti a partire da un IntStream.

TOOL VS CODE

Programmando in Java, è possibile fare click destro sull'IDE e cliccare sull'opzione **source action** del menù a tendina, da qui potremo generare getter/setter, costruttori, toString e altri metodi standard, senza doverli scrivere.

Slide Git
Slide Maven

DEBUG CON PEEK

Per esigenze di debug potrei voler conoscere come è fatto lo stream mano a mano che si eseguono le operazioni

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);

   numbers.stream()
           .map(x -> x + 17)
           .filter(x -> x % 2 == 0)
           .limit(3)
           .forEach(System.out::println);
           // Output: 20 22
```

Sarebbe utile capire cosa produce ciascuna operazione. Il metodo forEach() consuma l'elemento dello stream, quindi non si può usare

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);
   numbers.stream()
   .peek(x -> System.out.println("from stream: " + x))
   .map(x -> x + 17)
   .peek(x -> System.out.println("after map: " + x))
   .filter(x -> x % 2 == 0)
   .peek(x -> System.out.println("after filter: " + x))
   .limit(3)
   .peek(x -> System.out.println("after limit: " + x))
   .collect(Collectors.toList());
    /* Output:
   from stream: 2
   after map: 19
   from stream: 3
   after map: 20
   after filter: 20
   after limit: 20
   from stream: 4
   after map: 21
   from stream: 5
   after map: 22
   after filter: 22
   after limit: 22 */
```

Per risolvere il problema utilizziamo **peek(),** un'operazione non terminale che ci permette di debuggare il codice.

RIEPILOGO METODI STREAM

metodo	parametri in input	output	tipo operazione
filter	Predicate	Stream	intermedia
count		long	terminale
of	lista di valori	Stream	intermedia
reduce	valore iniziale, e operatore binario	valore	terminale
map	funzione	Stream	intermedia
max	Comparator	Optional	terminale
collect	Collector	collezione	terminale

Riepilogo Su Stream (2)

- Metodi definiti da Stream, visti in questa lezione

metodo	parametri in input	output	tipo operazione
sorted	Comparator	Stream	intermedia
forEach	Consumer	void	terminale
distinct		Stream	intermedia
findAny		Optional	terminale
findFirst		Optional	terminale
iterate	seme e funzione	Stream	intermedia
limit	dimensione	Stream	intermedia
generate	Supplier	Stream	intermedia
mapToInt	Function	IntStream	intermedia
peek	Consumer	Stream	intermedia

- Metodi definiti da IntStream

rangeClosed	valore iniziale, valore finale	IntStream	intermedia
sum		int	terminale
boxed		Stream	intermedia
mapToObj	Function	Stream	intermedia