LAMBDA EXPRESSION

Le lambda expression in Java sono utilizzate per creare delle **funzioni anonime**:

- Un'espressione lambda è una funzione senza dichiarazione (che non ha nome)
- Un'espressione lambda può anche essere considerata come un nuovo tipo di dato

Un'espressione lambda può:

- Essere passata come argomento in un metodo
- Essere restituita in uscita da un metodo

Le lambda expr hanno particolare utilità nei casi in cui serve definire una funzione con poche linee di codice, che verrà utilizzata una sola volta [Si evita di implementare un metodo classico]

Esempio:

```
public class EsempiLambdaExpression {
  public static void main(String[] args) {
    Thread t1 = new Thread() {
      @Override
      public void run() {
        System.out.println("Ecco un thread creato senza Lambda Expressions");
      }
    };

    Thread t2 = new Thread(() -> System.out.println("Questo è un thread creato usando le Lambda Expressions...!"));

    t1.start();
    t2.start();
  }
}
```

Le espressioni lambda hanno migliorato notevolmente la gestione delle liste, rendendo più semplice le operazioni di:

- Iterazione
- Ricerca
- Estrazione dei dati

INTERFACCE FUNZIONALI

Una functional interface è un'interfaccia che definisce un solo metodo ed è identificata mediante la annotazione **@FunctionalInterface**

- Una functional interface può essere implementata da una espressione lambda che effettua l'override del suo metodo (funzione anonima, che definisce l'unico metodo della IF)

```
@FunctionalInterface
public interface Test{ public void prova(); }
```

Prima di Java8:	Con Java8:
• Interfaccia con metodo da implementare • Classe Concreta che implementa l'interfaccia e metodo • Creazione istanza della classe concreta che usa il metodo	• Si crea Functional Interface • Non si ha bisogno di implementare una classe che implementi la IF: si utilizza direttamente la IF creando al volo un oggetto di un nuovo tipo

	○ La lambda expr fornisce una implementazione concreta del metodo • Si può direttamente usare l'oggetto

Prima di Java 8…

Creo un'interfaccia FormaGeometrica che definisce il metodo calcolaArea(…);

```
public interface FormaGeometrica {
   public double calcolaArea(double lato1, double lato2);
}
```

Creo una classe che implementa la nostra interfaccia…

```
public class Rettangolo implements FormaGeometrica {
  @Override
  public double calcolaArea(double lato1, double lato2) {
    return lato1*lato2;
  }
}
```

A questo punto, posso utilizzare la nuova classe per creare un oggetto di tipo FormaGeometrica

```
FormaGeometrica r1 = new Rettangolo();
r1.calcolaArea(3, 4);
```

Con Java 8…

Creo un'interfaccia FormaGeometrica che definisce il metodo calcolaArea(…); specifico che si tratta di una Functional Interface per agevolare la JVM…

```
@FunctionalInterface
public interface FormaGeometrica {
   public double calcolaArea(double lato1, double lato2);
}
```

Non ho bisogno di creare nessuna classe implementi la nostra interfaccia…

Posso utilizzare l'interfaccia funzionale creando «al volo» il tipo Rettangolo, implementando, attraverso una espressione lambda, l'unico metodo definito nell'interfaccia funzionale FormaGeometrica.

```
FormaGeometrica Rettangolo = (a, b) -> a * b;
Rettangolo.calcolaArea(3, 4);
```

SINTASSI DELLE LAMBDA EXPRESSION

`( Lista argomenti ) -> { istruzioni; }`	`( Lista argomenti ) -> espressione`

Esempi:

`()->System,out.println("");`	Funzione che non riceve argomenti in ingresso e stampa qualcosa
`(int a, int b) -> a*b;`	Prende in ingresso due numeri e ne restituisce il prodotto

(String nome)->nome.toUpperCase()	Riceve una stringa e la restituisce con i caratteri maiuscoli
(String nome, String cognome) -> {Stringa res = nome+" "+cognome; sysout(res);}	Sequenza di più istruzioni

Le istruzioni **break** e **continue:**

- non si possono utilizzare direttamente nel blocco di istruzioni della lambda ({…})
- si possono utilizzare all'interno di un ciclo contenuto all'interno del blocco {…}

Espressioni lambda:

- si può fare in modo che restituiscano un valore: si usa **return**
- si può omettere il tipo dei parametri in ingresso
- omettere la parentesi per la lista degli argomenti, se c'è un solo parametro

NOTA:

Le funzioni lambda sono utilizzate, quando si ha a che fare con **funzioni di ordine superiore**, che sono quelle funzioni che prendono in ingresso un'altra funzione.

- Sono usate per ordinare liste, collezioni, eseguire task pianificati
- Prima si usavano classi dedicate (es: Comparator, per creare la classe per comparare gli elementi di una lista)

Esempio:

Runnable è una **interfaccia funzionale:** definisce soltanto il metodo ***void run()*** che non riceve parametri e fa qualcosa**.**
Un generico Thread riceve in ingresso un Runnable all'interno del suo costruttore; si possono usare le lambda expr, in modo tale che invece di istanziare una classe ad hoc, il Runnable da usare lo si crei al volo

Thread t = new Thread(<runnable>); --- runnable: @Override public void run()

- Definisco una f anonima che fornisca un'implementazione del metodo dell'interfaccia funzionale: tale implementazione sarà usata per creare un oggetto che abbia come tipo quello della IF**.**

Thread t = new Thread(()->{task da eseguire nel metodo run});

Comparator è una **interfaccia funzionale;** essa definisce il metodo
int compare(Object o1, Object o2)

Il Comparator è usato, ad esempio, come parametro in ingresso al metodo sort di Collections: lo si può implementare al volo**.**

```
LinkedList<String> ss = new LinkedList<>();
ss.add("…
Collections.sort(ss, (String s1, String s2)->{
                  if(s1.length()>s2.length()) { return 1; }
                  else {
                        if(s1.length()==s2.length()) {return 0;}
                        else {return -1;}
                        }
                  }
            );
```

Riassumendo:

- Le lambda expression sono usate per definire in maniera anonima (al volo), l'**implementazione del metodo di una interfaccia funzionale**
- In particolare si possono scrivere come:
  - **parametro passato ad un metodo**:
    - Se un metodo ha bisogno di ricevere un'istanza di una interfaccia funzionale, anziché definire un oggetto che implementi la IF, si passa una lambda expr, che "crea l'istanza" della IF, definendo il metodo che essa espone.
    - Non si passa direttamente un'istanza della IF, ma si passa l'implementazione del metodo che la IF espone; esso sarà usato dietro le quinte per creare al volo un'istanza di oggetto, che implementa la IF

```
Collections.sort(list,(o1,o2)->{…});  //COMPARATOR
```

  - **Assegnamento**:
    - Si crea un'istanza della IF funzionale, assegnandole mediante lambda expr la definizione del metodo che espone

```
Runnable p = ()->{…};  //IMPLEMENTAZIONE run()
Thread t = new Thread(p);
```

- A livello di sintassi, è come se stessi definendo un metodo:
  - **`(lista param) -> {istruzioni;}`**
    - Gli attributi della lista dei parametri, essendo appunto variabili, si possono chiamare con un nome a piacere
    - Si può omettere il tipo: viene dedotto dall'ordine con cui i parametri sono passati (perché si sta ridefinendo l'unico metodo dell'interfaccia funzionale)

PACKAGE java.util.function

È un package che contiene una serie di interface funzionali, pronte per essere implementate mediante le lambda expr.

PREDICATE<T>	Interfaccia funzionale che rappresenta un "predicato" di un elemento.
	▪ **Predicato**: proposizione che contiene delle variabili, il cui valore determina la veridicità della proposizione
	▪ **Il predicato può essere usato come filtro per la ricerca: a seconda del come si implementa il metodo test si può cambiare il metodo di ricerca**
	
	Metodo:
	`boolean test(T t);`
	
	Utilizzo:
	▪ Nella classe d'interesse si definisce il metodo che riceve il predicato come parametro e lo utilizza per svolgere la computazione
	▪ Quando si richiama la funzione, si definisce "al volo" il predicato, mediante la lambda expr
	
	`public static boolean maggiore(String s, Predicate<String> p)`
	`{ return p.test(s); }`
	
	`maggiore("TST", (String str)->{return str.length()>2;}));`
	`maggiore("TST", (String str)->{return str.length()>3;}));`
	`maggiore("TST", (String str)->{return str.length()>4;}));`

	VANTAGGIO: Il metodo restituisce true ad un match; il criterio del come eseguire questo match, può essere ridefinito all'occorrenza, direttamente nella lambda. Lo stesso metodo può essere usato per eseguire ricerche con filtri diversi.

CONSUMER<T>	Interfaccia funzionale che implementa un'operazione che accetta parametri in ingresso e non ritorna valori
	▪ **Si usa per eseguire operazioni che non producono risultati**
	
	Metodo:
	void accept(T t);
	
	Utilizzo:
	▪ Nella classe d'interesse si definisce il metodo che riceve il consumer
	▪ Il consumer eseguirà l'operazione sul parametro che riceve
	
	`List<String> list = …`
	`Consumer<String> cons =`
	`(String elem)->`
	`{System.out.println(elem);};`
	`list.forEach(c);`
	`/**----- OPPURE -----**/`
	`for(String s:list){cons.accept(s);}`
	
	NOTA:
	le liste hanno definito un metodo **forEach(Consumer<T>)** che accetta un Consumer per eseguire le operazioni sui singoli elementi presenti nella lista.
	
	VANTAGGIO: L'implementazione del codice da eseguire su di un elemento, può variare a seconda di ciò che gli si passa.

SUPPLIER<T>	Interfaccia funzionale che implementa un'operazione che non accetta parametri in ingresso e ritorna un valore
	▪ **Si usa per eseguire operazioni che non producono risultati**
	
	Metodo:
	T get();
	
	Utilizzo:
	▪ Nella classe d'interesse si definisce il metodo che riceve il Supplier
	▪ Il supplier sfrutterà il metodo get() per restituire l'elemento sul quale eseguire l'operazione
	
	`List<String> list=…`
	`for (String str : list) {`
	`stampa (() -> str); //LA GET() DEL SUPPLIER RICEVE LA STRINGA`

	}
	
	private void stampa (Supplier<String> supp) {
	  System.out.println(sup.get());
	}
	
	▪ Itero sugli elementi della lista
	▪ Per ogni elementi richiamo il metodo stampa
	▪ Stampa riceve un ingresso un supplier
	▪ Definisco il supplier con la lambda: definisco il suo metodo get() in modo che restituisca la stringa i-ma dell'iterazione
	▪ Nel metodo stampa uso la stringa tramite la get()

FUNCTION<T,R>	Interfaccia funzionale che implementa una generica funzione: riceve in ingresso un argomento T e produce/restituisce un risultato R
	▪ **Si usa per eseguire operazioni che producono un risultato da restituire**
	▪ **È utilizzata quando si ha la necessità di passare in input ad un metodo un blocco di codice**
	
	Metodo:
	R apply(T t);
	Utilizzo:
	▪ Si definisce un metodo generico che riceve in ingresso una Function<T,R>
	▪ La Function si definisce con una lambda expr che ridefinisce il metodo apply (quello che deve eseguire)
	▪ A seconda del tipo di funzione passata, il metodo opererà cose diverse
	
	operazione(3,**(Double val)->{return val*val;}**);
	operazione(3,**(Double val)->{return val+val;}**);
	
	public static double operazione(double a,
	Function<Double,Double>
	f) {
	  return f.apply(a);
	}
	
	VANTAGGIO: L'implementazione dell'operazione da eseguire su di un elemento, può variare a seconda di ciò che gli si passa alla funzione.

UNARYOPERATOR<T>	Interfaccia funzionale che implementa una generica operazione, che riceve in ingresso un operando T e produce/restituisce un risultato dello stesso tipo T ▪ **Si usa quando si ha la necessità di effettuare un'operazione su un dato, che lo modifichi (o che restituisca stesso tipo)** Metodo: **T apply(T t);** Utilizzo: ▪ Si definisce un metodo generico che riceve in ingresso un UnaryOperator<T>

	▪ Lo UnaryOperator<T> è implementato mediante la definizione della sua funzione apply, mediante lambda expr
	▪ A seconda del tipo di funzione passata, il metodo opererà cose diverse
	
	`UnaryOperator<String> op =`
	`(str)-> {return str.toUpperCase();};`
	`System.out.println(op.apply("string");`
	
	`UnaryOperator<Double> op =`
	`(d)-> {return d*d;};`
	`System.out.println(op.apply(10.0);`

BINARYOPERATOR<T>	Interfaccia funzionale che implementa una generica operazione su due operandi dello stesso tipo T, che produce un risultato dello stesso tipo degli operandi
	▪ **Si usa quando si ha la necessità di effettuare un'operazione su due operandi**
	Metodo:
	`T apply(T t1, T t2);`
	
	Utilizzo:
	▪ Si definisce un metodo generico che riceve in ingresso un BinaryOperator<T>
	▪ Il BinaryOperator <T> è implementato mediante la definizione della sua funzione apply, mediante lambda expr
	▪ A seconda del tipo di funzione passata, il metodo opererà cose diverse
	
	`BinaryOperator<String> op = (a,b)->{return a+b;};`
	`System.out.println(op.apply("a","b"));`

FILTRARE GLI ELEMENTI DI UNA LISTA USANDO LE LAMBDA EXPR

- Ricerca di uno o più elementi (con uno o più criteri di ricerca)

```
public List<String> cercaStringhe
  (List<String> elenco,
       Predicate<String> filter){
List<String> trovati =
               new ArrayList<>();
for(String s:elenco) {
      if(filter.test(s)) {
            trovati.add(s);
      }
}
return trovati;
}


cercaStringhe(lista, ()->{…})
```

```
public String cercaStringa
      (List<String> elenco,
           Predicate<String> filter){
 for(String s:elenco) {
      if(filter.test(s)) {
            return s;
      }
}
      return null;
}




cercaStringa(lista, ()->{…})
```

STREAM

Strumento che consente di processare gli oggetti contenuti in una Collection.

- ***Stream***: sequenza di oggetti ottenuti da una specifica sorgente, sui quali è possibile compiere delle operazioni
    - Consente l'accesso in modo sequenziale ad un insieme di elementi di un tipo specifico;
    - Gli elementi dello stream possono essere recuperati da: una collezione, un array o una operazione di I/O;
    - Supporta operazioni di aggregazione
    - Molte operazioni sugli stream restituiscono stream, quindi possono essere concatenate.

STREAM PROCESSING:	
stream()	crea il flusso di oggetti da analizzare

Configurazione	
filter(Predicate<T>)	filtra gli elementi, usando un predicato
map(Function<T>)	Mappa/trasforma gli elementi in ingresso, in altri elementi in uscita
limit(int)	fissa il limite max del n° di elementi
sorted(Comparator<T>)	Ordina gli elementi dello stream
distinct()	Restituisce solo elementi distinti

Elaborazione	
collect(Collector<T,R,A>)	restituisce lista o risultati di elaborazioni parziali fatte sugli elementi
max/min(Comparator<T>)	valori min/max presenti nello stream
count()	restituisce il numero di elementi nello stream
reduce(BinaryOperator<T>)	mette gli elementi di uno stream in un singolo oggetto

---

OTTENERE UNO STREAM

metodo di Collection:

- **nomeCollezione.stream()**

```
List<String> items = new ArrayList<String>();
items.add("uno");
items.add("tre");
items.add("otto");
items.add("undici");
Stream<String> stream = items.stream();
```

ELABORAZIONE DEGLI ELEMENTI

1. **Configurazione:** processo di filtraggio/mappatura degli elementi

2. **Elaborazione:** esecuzione di un'operazione specifica sugli elementi filtrati o mappati

CONFIGURAZIONE

FILTRAGGIO DATI

Filtra della collezione i dati che soddisfano una proprietà

- **stream.filter(Predicate<T>)**

```
Stream<String> filteredStream = list.stream().filter(new Predicate<String>() {
    //DEFINIZIONE PREDICATO: SI PUO' PASSARE LA LAMBDA EXPR
    @Override
    public boolean test(String str) {…}
});
```

MAPPING DATI

Mappa gli oggetti della collezione, in altri da essi derivati

- **stream.map(Function<T>)**

```
Stream<Integer> mappedStream = list.stream().map(new
                                        Function<String,Integer>() {
    @Override
    public Integer apply(String str) {…}
});
```

LIMIT

Riduce la dimensione degli elementi dello stream, ad un numero specificato, eliminando quelli in eccesso

- **stream.limit(int)**

```
Stream<String> limitedStream = list.stream().limit( 2 );
```

SORTED

Ordina gli elementi dello stream:

- se non si specifica nessun parametro, si seguirà l'ordinamento naturale degli elementi dello stream
- per imporre un criterio di ordinamento, riceve un Comparator<T>

- **stream.sorted(Comparator<T>)**

```
Stream<String> sordedStream = list.stream().sorted( new Comparator<String>() {
    @Override
    public int compare(String item1, String item2) {
        return Integer.valueOf( item1.length() )
                      .compareTo( Integer.valueOf( item2.length() ) );
    }
});

.sorted(Comparator.reverseOrder())

.sorted(Comparator.comparing(Student::getFirstName)
                  .thenComparing(Student::getLastName)).map(Student::getName)
```

DISTINCT

Restituisce uno stream contenente solo elementi diversi

- **stream.distinct()**

ELABORAZIONE

COLLECT

Riceve in input un oggetto Collector, utilizzato per mettere (collezionare) gli elementi di uno stream, in una nuova struttura o effettuare su di essi altre operazioni.
È principalmente usato per:

- inserire gli elementi in una nuova lista;
- inserire gli elementi in un set;
- convertire gli elementi in stringhe e concatenazione;
- raggruppare gli elementi;
- calcolare la somma dei valori assunti da una proprietà degli elementi;
- partizionare gli elementi;

Collectors espone dei metodi statici, che restituiscono i Collector più utilizzati

- **stream.collect(Collector<? Super T,A,R>)**
  - **T = input, A = tipo intermedio di accumulazione, R = tipo del risultato**
- **Stream.collect(Collectors.metodoRestituisceCollectUtile())**

```
// Inserimento in una nuova lista
list.stream().collect(Collectors.toList());

// Inserimento in una mappa
list.stream().collect(Collectors.toSet());

// Concatenamento
list.stream().collect(Collectors.joining(","));
```

Esempi applicativi di collect e Collector disponibili da Collectors

Esempio#1: Restituisco lista

```
List<String> items = new ArrayList<String>();
List<Integer> l = items.stream().map((String e)->{return 1;}).
                                                collect(Collectors.toList());
finale.forEach((e)->System.out.println(e));
```

#1.1:

```
List<String> list = people.stream().map(Person::getName)
                                   .collect(Collectors.toList());
```

Esempio#2: Restituisco TreeSet

```
Set<String> set = people.stream().map(Person::getName)
                        .collect(Collectors.toCollection(TreeSet::new));
```

Esempio#3: Converto la lista in stringhe e li unisco in un'unica stringa

```
String joined = things.stream().map(Object::toString)
                               .collect(Collectors.joining(", "));
```

Esempio#4: Sommo un parametro di tutti gli oggetti della lista

```
int total = employees.stream()
                        .collect(Collectors.summingInt(Employee::getSalary)));
```

Esempio#5: Raggruppo gli oggetti della lista secondo una proprietà

```
Map<Department, List<Employee>> byDept = employees.stream()
                     .collect(Collectors.groupingBy(Employee::getDepartment));
```

Esempio#6: Raggruppo gli oggetti della lista secondo una proprietà e ne faccio la somma di un altro parametro

```
Map<Department, Integer> totalByDept = employees.stream()
                    .collect(Collectors.groupingBy(Employee::getDepartment,
                                    Collectors.summingInt(Employee::getSalary)));
```

Esempio#6: Partiziono gli elementi della lista in funzione di un test booleano

```
Map<Boolean, List<Student>> passingFailing = students.stream()
      .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

[partitioningBy(Predicate<T>)]

MIN/MAX

Restituiscono i valori min e max presenti nello stream: richiedono in input un Comparator e restituiscono un Optional (che è un oggetto contenitore)

- **stream.max(Comparator<T>)**

```
mappedStream.max( new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo( o2 );
    }
});
```

COUNT

Restituisce il numero di elementi nello stream, dopo che è stato filtrato

- **stream.count()**

```
filteredStream.count();
```

REDUCE

Riduce gli elementi di uno stream in un singolo oggetto. Richiede come parametro un oggetto che implementa BinaryOperator<T>.

- il metodo apply(T,T) del BinaryOperator, prevede in input un parametro accumulatore ed un item dello stream e restituisce un **Optional**

- **stream.reduce(BinaryOperator<T>)**

```
filteredStream.reduce( new BinaryOperator<Integer>() {
    @Override
    public Integer apply(Integer acc, Integer item) {
        return acc + item
    }
});
```

OPTIONAL

È un tipo introdotto per *rappresentare oggetti che possono assumere valori nulli*.

Ne esistono specializzazioni per rappresentare alcuni tipi primitivi: OptionalDouble, OptionalInt e OptionalLong.

Un oggetto Optional può trovarsi i due stati possibili:

- **PRESENT**: ovvero contiene un riferimento non nullo ad un oggetto di tipo T;
- **ABSENT** or **EMPTY**: in caso opposto.

#NOTA:

Un oggetto Optional empty non è equivalente ad un oggetto nullo.

Per instanziare un nuovo oggetto di tipo Optional esistono diverse possibilità:

- `Optional.of()`: metodo statico che crea un Optional per un oggetto non nullo, altrimenti solleva l'eccezione NullPointerException.
- `Optional.ofNullable()`: metodo statico che crea un Optional per un oggetto che può essere nullo, nel caso restituendo un oggetto Optional.empty().
- `Optional.empty()`: metodo statico che restituisce un oggetto Optional empty.

`Optional<T> op = …` `op.get()`	Restituisce l'oggetto T referenziato in op. Se invocato su di un Optional vuoto, solleva una eccezione
`Optional<T> op = …` `op.isPresent()`	Restituisce true se l'oggetto referenziato in op non è vuoto

findAny() e findFirst(): metodi che generano Optional

stream().filter().metodiOptional() **NOTA:** i metodi or interrogano l'Optional e vedono se è vuoto	
findAny()	Restituisce un **Optional** che contenga un elemento scelto non deterministicamente.
findFirst()	Restituisce un **Optional** che contiene il primo elemento dello stream.
findAny().orElse(T)	Restituisce il valore dell'elemento dello stream se presente, oppure l'oggetto T passato come argomento [interroga l'Optional restituito da findAny()]
findAny() .orElseGet(Supplier<? extends T>))	Restituisce il valore se presente, oppure invoca il Supplier, restituendo il risultato che produce
findAny() .orElseThrow(Supplier<? extends T>))	Restituisce il valore dello stream filtrato se presente, oppure, solleva l'eccezione fornita
findAny().ifPresent (Consumer<T>)	Se è presente un valore nello stream filtrate, invoca lo specific consumer sul valore; in caso contrario non fare niente.

ESEMPI: