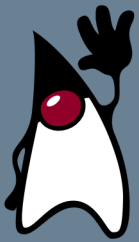




UNIVERSITÀ DEGLI STUDI DELL'AQUILA

# Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco  
[juri.dirocco@univaq.it](mailto:juri.dirocco@univaq.it)  
<https://jdirocco.github.io>





# Sommario

---

- › Nested class
  - Inner class
  - Classi anonime
  
- › Espressioni Lambda
  - Cosa è
  - Visibilità
  - @FunctionalInterface
  - Gestione eccezioni
  - Espressioni Lambda e collections
  - Esempi



# Nested class

---

- › Java è possibile annidare definizione di classi all'interno di altre classi (nested class o classi innestate)
- › In caso di classi che NON sono dichiarate `static` si parla di classi interne (o inner class)
- › Permette di raggruppare classi logicamente correlate che non ha senso rendere visibili all'esterno
- › Particolarmente utili nella realizzazione di interfacce grafiche (GUI)
- › Sintassi

```
class <nome classe> {  
    class <nome classe interna> {  
    }  
}
```



# Inner class: Esempio (1)

---

```
public class Outer {  
    private String messaggio = "Nella classe ";  
    private void stampaMessaggio() {  
        System.out.println(messaggio + "Esterna");  
    }  
    /* la classe interna accede in maniera naturale ai membri della classe  
       che la contiene */  
    public class Inner {  
        // classe interna  
        public void metodo() {  
            System.out.println(messaggio + "Interna");  
        }  
        public void chiamaMetodo() {  
            stampaMessaggio();  
        }  
    }  
}
```

Vengono creati due files: Outer.class e Outer\$Inner.class



## Inner Class (2)

---

- › Classe interna può accedere ai membri privati della classe esterna
- › E' possibile costruire oggetti della classe `Inner` fuori dalla classe `Outer`

```
Outer outer = new Outer();
```

```
Outer.Inner inner = outer.new Inner();
```

- › Si possono utilizzare i modificatori di accesso `private` e `protected` nella definizione della inner class



## Inner Class (3)

```
public class Outer2 {  
    private String stringa = "esterna";  
    public class Inner2 {  
        private String stringa = "interna";  
        public void metodoInner() {  
            System.out.println(Outer2.this.stringa +  
                " " + this.stringa);  
        }  
    }  
}  
public static void main(String [] args) {  
    Outer2 outer = new Outer2();  
    Outer2.Inner2 inner = outer.new Inner2();  
    inner.metodoInner();  
}
```

Riferimento alla variabile di istanza



# Classi anonime (1)

---

- › Inner Classi senza nome
- › Utilizzate generalmente per gestire gli eventi in interfacce grafiche (GUI)
  - Esempio click di un mouse
- › Classe anonima richiede che
  - Contestualmente alla dichiarazione della classe venga istanziato un suo oggetto
  - Esistenza **di una sua superclasse o di una sua super interfaccia** di cui sfrutterà il costruttore (solo virtualmente nel caso di un'interfaccia)
  - Non può avere un costruttore



## Classi anonime (2)

### › Esempio

```
public class ClasseEsistente {  
    public void metodo() {  
        System.out.println("Nella classe esistente");  
    }  
}  
  
public class Outer4 {  
    private String messaggio = "Nella classe ";  
    //Definizione della classe anonima e sua istanza  
    ClasseEsistente ce = new ClasseEsistente() {  
        @Override  
        public void metodo() {  
            System.out.println(messaggio + "anonima");  
        }  
    };  
}
```

Vengono creati due files: Outer4.class e Outer4\$1.class





## Classi anonime (3)

---

```
public static void main (String[] args) {  
    Outer out = new Outer();  
    out.ce.metodo();  
}
```



# Classi anonime (4)

## › Esempio alternativo

```
class ClasseEsistente {  
    public void metodo() {  
        System.out.println("Nella classe esistente");  
    }  
}  
  
class Outer4 {  
    private String messaggio = "Nella classe ";  
    ClasseEsistente ce = new MyClasseEsistente();  
  
    class MyClasseEsistente extends ClasseEsistente {  
        @Override  
        public void metodo() {  
            System.out.println(messaggio + "anonima");  
        }  
    };  
}
```



## Classi anonime (4)

---

```
ClasseEsistente ce = new ClasseEsistente() {  
    @Override  
    public void metodo() {  
        System.out.println(messaggio+"anonima numero "+ a);  
    }  
  
    public void metodoB() {  
        System.out.println("...");  
    }  
};  
  
ce.metodoB(); ???
```



## Classi anonime (5)

### › Esempio dentro metodo con utilizzo parametro formale

Non obbligatorio se non viene modificato parametro  
(effectively final)

```
public class Outer4 {  
    private String messaggio = "Nella classe ";  
    public void metodoConClasse(final int a) {  
        ClasseEsistente ce = new ClasseEsistente() {  
            @Override  
            public void metodo() {  
                System.out.println(messaggio + "anonima numero " + a);  
            }  
        } ;  
        ce.metodo();  
    }  
}
```



## Classi anonime (6)

### › Esempio dentro metodo con utilizzo parametro formale

```
public class Outer4 {  
    private String messaggio = "Nella classe ";  
    public void metodoConClasse(int a) {  
        a = 10;  
        ClasseEsistente ce = new ClasseEsistente() {  
            @Override  
            public void metodo() {  
                System.out.println(messaggio + "anonima numero " + a);  
            }  
        } ;  
        ce.metodo();  
    }  
}
```

Errore in compilazione



## Classi anonime (7)

---

### › Esempio con interfaccia

```
public interface Volante {  
    void plana();  
    void decolla();  
    void atterra();  
}
```

```
public class TestVolanteAnonymous {  
    public static void main(String[] args) {  
        .....  
    }
```



## Classi anonime (8)

---

```
Volante ufo = new Volante() {  
    @Override  
    public void decolla() {  
        System.out.println("Un oggetto non identificato sta decollando");  
    }  
    @Override  
    public void plana() {  
        System.out.println("Un oggetto non identificato sta planando");  
    }  
    @Override  
    public void atterra() {  
        System.out.println("Un oggetto non identificato atterra...");  
    }  
} ;  
ufo.decolla();  
ufo.plana();  
ufo.atterra();  
}
```



# Classi anonime (9)

## › Esempio 2 con interfaccia

```
public class Pilota {  
    private String nome;  
    public Pilota (String nome) {  
        this.nome = nome;  
    }  
    public void fattiUnGiro(Volante volante) {  
        volante.decolla();  
        volante.plana();  
        volante.atterra();  
    }  
    public void setName(String nome) {this.nome = nome;}  
    public String getNome() {return nome;}  
    @Override  
    public String toString(){return getNome();}  
}
```





# Classi anonime (10)

---

```
public class TestPilota {  
    public static void main(String args[]) {  
        Pilota pilota = new Pilota("Simone");  
        pilota.fattiUnGiro(new Volante() {  
            @Override  
            public void decolla() {  
                System.out.println("Un oggetto non identificato sta decollando");  
            }  
            @Override  
            public void plana() {  
                System.out.println("Un oggetto non identificato sta planando");  
            }  
            @Override  
            public void atterra() {  
                System.out.println("Un oggetto non identificato atterra...");  
            }  
        } );  
    }  
}
```



# Espressione lambda: cosa è (1)

---

- › Espressione lambda è detta anche **funzione anonima**
- › Funzione che non appartiene ad alcuna classe
- › Permettono di passare funzioni (codice) a dei metodi
- › Sintassi  
`([lista-parametri]) -> {blocco codice}`
- › I parametri possono essere vuoti
- › Possibile **omettere** il tipo dei parametri
- › Blocco codice sono le istruzioni da eseguire
  - Se è presente una sola istruzione le parentesi graffe possono essere omesse
  - Se blocco codice prevede una sola istruzione con `return` allora `return` si può omettere




# Espressione lambda: cosa è (2)

## › Interfaccia funzionale

- Interfaccia che ha un solo metodo astratto
- Può contenere metodi di default, statici, privati e costanti

## › Esempio

```
public class FirstLambda {  
    public static void main(String args[]) {  
        new Thread(new Runnable() {  Classe anonima  
            @Override  
            public void run() {  
                System.out.println("Prima di Java 8: Classe anonima");  
            }  
        } ).start();  
    }  
}
```




# Espressione lambda: cosa è (2)

## › Interfaccia funzionale

- Interfaccia che ha un solo metodo astratto
- Può contenere metodi di default, statici, privati e costanti

## › Esempio

```
public class FirstLambda {  
    public static void main(String args[]) {  
        new Thread(new Runnable() {  Classe anonima  
            @Override  
            public void run() {  
                System.out.println("Prima di Java 8: Classe anonima");  
            }  
        } ).start();  
        Runnable r = ()-> System.out.println("Java 8: Funzione anonima");  
        new Thread(()-> System.out.println("Java 8: Funzione anonima")).start();  
    }  
}
```



# Visibilità (1)

- › Classi anonima `this` si riferisce all'oggetto corrente istanziato dalla classe anonima
  - Per usare i membri della classe che include la classe anonima  
`NomeClasseEsterna.this.nomemembro`
- › Espressioni lambda `this` si riferisce direttamente alla classe in cui è inclusa l'espressione

```
public class LambdaThis {  
    private String stringa ="variabile d'istanza della classe";  
    public void metodoContenenteLambda() {  
        String stringa ="variabile locale del metodo contenente";  
        new Thread(()->System.out.println(this.stringa)).start();  
        new Thread(()->System.out.println(stringa)).start();  
    }  
    public static void main(String args[]) {  
        LambdaThis lambdaThis = new LambdaThis();  
        lambdaThis.metodoContenenteLambda();  
    }  
}
```



## Visibilità (2)

- › Variabili locali al blocco dell'espressione hanno un loro scope → non è possibile ridefinire la variabile

```
public void metodoContenenteLambda() {  
    String stringa = "variabile locale del metodo contenente";  
    new Thread(() -> {  
        String stringa = "variabile locale nell'espressione lambda";  
        System.out.println(stringa);  
    } ).start();  
}
```

```
amleto@LAPTOP-JNFQE190 MINGW64 ~/Desktop  
$ javac LambdaThis.java  
LambdaThis.java:6: error: variable stringa is already defined in method metodoContenenteLambda()  
        string stringa = "variabile locale nell'espressione lambda";  
        ^  
1 error
```



## Visibilità (3)

---

› E' possibile utilizzare variabili al di fuori dell'espressione purché sia final oppure *effettivamente non modificata*

› Esempio

```
public void startCount() {  
    int count = 0;  
    new Thread(() -> {  
        while (count < 100) {  
            System.out.println(count);  
        }  
    });  
}
```



## Visibilità (4)

### › Esempio

```
public void startCount() {  
    int count = 0;  
    new Thread(() -> {  
        while (count < 100) {  
            System.out.println(count++);  
        }  
    });  
}
```

```
amleto@LAPTOP-JNFQE190 MINGW64 ~/Desktop  
$ javac TestCount.java  
TestCount.java:9: error: local variables referenced from a lambda expression must be final or effectively final  
    while (count < 100) {  
           ^  
TestCount.java:10: error: local variables referenced from a lambda expression must be final or effectively final  
        System.out.println(count++);  
                           ^  
2 errors
```





# FunctionalInterface

---

- › Da Java 8 è sicuramente la possibilità di utilizzare le lambda expressions, associandole ad una functional interface.
- › Una functional interface è a tutti gli effetti una normale interfaccia. Quello che rende un'interfaccia una functional interface è una semplice caratteristica: **avere un solo metodo astratto**.
- › Il concetto è semplice: se un'interfaccia ha un solo metodo astratto, allora la possiamo definire functional interface.



# FunctionalInterface

```
public interface MyInterface {  
    void myMethod();  
    // unico metodo astratto  
}
```

```
public interface MyInterface {  
    void myMethod1();  
    // metodo astratto 1  
    void myMethod2();  
    // metodo astratto 2  
}
```

//Quests non è una  
FuntionalInterface



# FunctionalInterface

---

- › Nel conteggio dei metodi che concorrono a rendere o meno un'interfaccia una functional interface, non concorrono i metodi di default e statici
- › Nel conteggio dei metodi astratti che concorrono a rendere un'interfaccia una functional interface, i metodi della classe Object non concorrono a questo conteggio.



# @FunctionalInterface (1)

---

## › Esempio

```
public class Film {  
    private String nome;  
    private String genere;  
    private int mediaRecensioni;  
  
    public Film (String nome, String genere, int mediaRecensioni) {  
        this.nome = nome;  
        this.genere = genere;  
        this.mediaRecensioni = mediaRecensioni;  
    }  
    ...  
}
```



## @FunctionalInterface (2)

---

- › Per definire un'interfaccia funzionale si utilizza l'annotazione `@FunctionalInterface`

**@FunctionalInterface**

```
public interface FiltroFilm {  
    boolean filtra(Film film);  
}
```



## @FunctionalInterface (3)

---

```
public class Videoteca {  
    private Film[] films;  
    public Videoteca () {  
        films = new Film[10];  
        caricaFilms();  
    }  
    public void setFilms(Film[] films) {this.films = films;}  
    public Film[] getFilms() {return films;}  
    public Film[] getFilmFiltrati(FiltroFilm filtroFilm) {  
        Film [] filmFiltrati = new Film[10];  
        for (int i = 0, j= 0; i< 10;i++) {  
            if (filtroFilm.filtra(films[i])) {  
                filmFiltrati[j] = films[i];  
                j++;  
            }  
        }  
        return filmFiltrati;  
    }  
}
```

...



## @FunctionalInterface (4)

---

...

```
private void caricaFilms() {  
    films[0] = new Film("Il Signore degli anelli", "Fantasy", 5);  
    films[1] = new Film("Star Wars", "Fantascienza", 5);  
    films[2] = new Film("Avatar", "Fantascienza", 3);  
    films[3] = new Film("Blade Runner", "Fantascienza", 4);  
    films[4] = new Film("XMen", "Fantascienza", 5);  
    films[5] = new Film("The Avengers", "Fantasy", 4);  
    films[6] = new Film("Matrix", "Fantascienza", 5);  
    films[7] = new Film("Lanterna Verde", "Fantasy", 3);  
    films[8] = new Film("Forrest Gump", "Drammatico", 5);  
    films[9] = new Film("Indiana Jones", "Avventura", 3);  
}  
}
```



## @FunctionalInterface (5)

---

```
public class TestVideotecaConClasseAnonima {  
    public static void main(String args[]) {  
        Videoteca videoteca = new Videoteca();  
        System.out.println("Bei Film:");  
        Film[] beiFilms = videoteca.getFilmFiltrati(new FiltroFilm() {  
            @Override  
            public boolean filtra(Film film) {  
                return film.getMediaRecensioni() > 3;  
            }  
        });  
        stampaFilm(beiFilms);  
        System.out.println("\nFilm di Fantascienza:");  
    }  
}
```





## @FunctionalInterface (6)

---

```
Film[] filmDiFantascienza = videoteca.getFilmFiltrati(new FiltroFilm() {  
    @Override  
    public boolean filtra(Film film) {  
        return "Fantascienza".equals(film.getGenere());  
    }  
});  
stampaFilm(filmDiFantascienza);  
}  
  
private static void stampaFilm(Film [] films) {  
    for (Film film: films) {  
        if (film != null) {  
            System.out.println(film);  
        }  
    }  
}  
}
```



# Esempio Lambda

```
(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}
```

Diagram illustrating the components of a lambda expression:

- Argument List**: (int arg1, String arg2)
- Arrow token**: ->
- Body of lambda expression**: {System.out.println("Two arguments "+arg1+" and "+arg2);}

- › Il corpo di una lambda expression può contenere zero, uno o più statements.
- › Se esiste un singolo statement curly le parentesi graffe possono essere omesse ed il **tipo di ritorno** della funzione anonima è lo stesso del corpo della lambda expression.



## @FunctionalInterface (7)

---

```
public class TestVideotecaConLambda {  
    public static void main(String args[]) {  
        Videoteca videoteca = new Videoteca();  
        System.out.println("Bei Film:");  
        Film[] beiFilms = videoteca.getFilmFiltrati(  
            (Film film)-> film.getMediaRecensioni() >3);  
        stampaFilm(beiFilms);  
        System.out.println("\nFilm di Fantascienza:");  
        Film[] filmDiFantascienza = videoteca.getFilmFiltrati(  
            (Film film)->"Fantascienza".equals(film.getGenere()));  
        stampaFilm(filmDiFantascienza);  
        System.out.println("\nFilm che finiscono con s:");  
        Film[] filmCheFinisconoConS = videoteca.getFilmFiltrati(  
            (Film film)->film.getNome().endsWith("s"));  
        stampaFilm(filmCheFinisconoConS);  
    }  
}
```



# Lambda calculus con più parametri (1)

---

```
public class TestCompleto
{
    // Inner interface
    interface FuncInter1 {
        int operation(int a, int b);
    }

    // sayMessage() is implemented using lambda expressions
    interface FuncInter2 {
        void sayMessage(String message);
    }

    private int operate(int a, int b, FuncInter1 fobj) {
        return fobj.operation(a, b);
    }

    ...
}
```



```
public static void main(String args[])    {
    FuncInter1 add = (int x, int y) -> x + y;
    FuncInter1 multiply = (int x, int y) -> x * y;
    TestCompleto tobj = new TestCompleto();
    System.out.println("Addition is " + tobj.operate(6, 3, add));
    System.out.println("Multiplication is " +
        tobj.operate(6, 3, multiply));

    FuncInter2 fobj = message ->System.out.println("Hello "
        + message);

    fobj.sayMessage("Geek");
}
}
```



# Gestione Eccezioni (1)

---

- › Problema nasce quando ci sono dichiarate le eccezioni **checked** nella dichiarazione del metodo dell'interfaccia funzionale
- › Espressioni lambda non hanno dichiarazione di metodo pertanto è un problema
- › Esempio

```
new Thread(() -> {  
    Thread.sleep(1000); //lancia eccezione InterruptedException  
    System.out.println("Hello World");  
} ).start();
```



# Gestione Eccezioni (2)

---

## › Soluzione

```
new Thread(() -> {  
    try {  
        Thread.sleep(1000);  
    }  
    catch (InterruptedException exc) {  
        exc.printStackTrace();  
    }  
    System.out.println("Hello World");  
} ).start();
```



## Gestione Eccezioni (3)

---

- › Se il metodo astratto dell'interfaccia funzionale lancia un'eccezione checked il codice dell'espressione lambda può lanciare l'eccezione checked o sotto-eccezioni

- › Esempio

```
import java.util.concurrent.*;

@FunctionalInterface
interface MyCallable extends Callable<Void> {

    @Override
    Void call() throws InterruptedException;

}

@FunctionalInterface
public interface Callable<V> {

    V call() throws Exception;

}
```





## Gestione Eccezioni (4)

---

```
public class TestMyCallable {  
    public static void main(String args[]) {  
        MyCallable callable = () -> {  
            Thread.sleep(1000);  
            System.out.println("Hello World");  
            return null; // necessario perché Void è un classe  
        } ;  
        ExecutorService pool = Executors.newFixedThreadPool(1);  
        pool.submit(callable); //<T> Future<T> submit(Callable<T> task)  
        pool.shutdown();  
    }  
}
```



## Gestione Eccezioni (5)

---

### › Senza interfaccia MyCallable

```
import java.util.concurrent.*;

public class TestWithoutMyCallable {
    public static void main(String args[]) {
        Callable<Void> callable = () -> {
            Thread.sleep(1000);
            System.out.println("Hello World");
            return null;
        };
        ExecutorService pool = Executors.newFixedThreadPool(1);
        pool.submit(callable);
        pool.shutdown();
    }
}
```



# Lambda Calculus e collection - forEach

---

```
class Test
{
    public static void main(String args[])
    {
        // Creating an ArrayList with elements
        // {1, 2, 3, 4}
        ArrayList<Integer> arrL = new ArrayList<Integer>();
        arrL.add(1); arrL.add(2); arrL.add(3); arrL.add(4);
        // Using lambda expression to print all elements of arrL
        arrL.forEach(n -> System.out.println(n));
        // Using lambda expression to print even elements of arrL
        arrL.forEach(n -> { if (n%2 == 0) System.out.println(n); });
    }
}
```



# Labda calculus and collection - sort

```
public class Demo {  
    public static void main(String[] args)  
    {  
        ArrayList<Integer> al = new ArrayList<Integer>();  
        al.add(205); al.add(102); al.add(98); al.add(275); al.add(203);  
        System.out.println("Elements of the ArrayList " +  
                            "before sorting : " + al);  
  
        // using lambda expression in place of comparator object  
        Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);  
        System.out.println("Elements of the ArrayList after sorting : " + al);  
    }  
}
```

› RICORDATE L'INTERFACCIA COMAPARATOR –

[https://download.java.net/java/early\\_access/panama/docs/api/java.base/java/util/Comparator.html](https://download.java.net/java/early_access/panama/docs/api/java.base/java/util/Comparator.html)



# Convertire da ArrayList a HashMap con lambda (1)

---

```
class Items {  
    private Integer key;  
    private String value;  
    public ListItems(Integer id, String name)    {  
        // assigning the value of key and value  
        this.key = id;  
        this.value = name;  
    }  
    public Integer getkey() { return key; }  
  
    // return private variable name  
    public String getvalue() { return value; }  
}
```



# Convertire da ArrayList a HashMap con lambda (2)

```
class Main {  
    public static void main(String[] args)    {  
        List<ListItems> list = new ArrayList<ListItems>();  
  
        list.add(new ListItems(1, "I"));  
        list.add(new ListItems(2, "Love"));  
        //...  
        list.add(new ListItems(5, "Geeks"));  
        Map<Integer, String> map = new HashMap<>();  
  
        list.forEach(  
            (n) -> { map.put(n.getKey(), n.getValue()); });  
  
        System.out.println("Map : " + map);  
    }  
}
```



# Uso di Stream nelle collections (1)

---

- › Una interessante caratteristica di Java 8 è la possibilità di processare gli oggetti contenuti in una collection attraverso l'utilizzo di un nuovo strumento: gli `Stream`.
- › non ha nulla a che vedere con gli stream utilizzati nella gestione dell'I/O
- › sono simili agli `Iterator` ma, diversamente, hanno lo scopo di consentire di elaborare gli oggetti della collezione utilizzando un approccio **dichiarativo**.
- › Uno stream rappresenta una sequenza di oggetti ottenuti da una specifica sorgente ai quali possiamo applicare una sequenza di 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, da un array o da una operazione di I/O;
  - Lo stream supporta operazioni di aggregazione (che vedremo nel seguito);
  - Molte operazioni sugli stream restituiscono stream, quindi possono essere concatenate.



## Uso di Stream nelle collections (2)

---

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





# Operazioni sullo Stream(1)

---

- › **Filtraggio:** Per filtrare lo stream è sufficiente utilizzare il metodo `filter()` che riceve in ingresso un oggetto che implementa l'interfaccia `java.util.function.Predicate` che definisce un solo metodo con firma `boolean test(T t)`, che riceve in ingresso un item della collezione e determina se filtrarlo o meno.
- › **Mapping:** mappare gli oggetti della collezione in altri oggetti da essi derivati. Allo scopo l'oggetto Stream espone il metodo `map()` che riceve in ingresso un oggetto che implementa l'interfaccia `java.util.function.Function`.
- › **Limit:** Consente di ridurre la dimensione dello stream ad un valore massimo specificato eliminando gli item in eccesso.
- › **Sorted:** Il metodo consiste di ordinare lo stream. Può essere utilizzato senza parametri e quindi l'ordinamento sarà quello naturale associato al tipo di elementi dello stream, oppure può accettare un oggetto di tipo `Comparator`.



## Operazioni sullo Stream(2)

---

- › **Collect**: La principale operazione di elaborazione è la `collect()`, che riceve in input un oggetto **Collector**. Tali oggetti sono utilizzati per combinare gli elementi appartenenti ad uno stream. Alcuni esempi di operazioni che possono essere eseguite sono:
- inserimento degli elementi in una nuova lista;
  - inserimento degli elementi in un set;
  - conversione degli elementi in stringhe e concatenazione;
  - raggruppamento degli elementi;
  - calcolo della somma dei valori assunti da una proprietà degli elementi;
  - partizionamento degli elementi;

Fortunatamente l'oggetto **Collectors** espone metodi statici che restituiscono molti dei Collector più utilizzati, fermo restando la possibilità per l'utente di implementare il proprio collector



## Operazioni sullo Stream(2)

---

- › **Min e max:** tali operazioni restituiscono i valore *minimo* o *massimo* tra gli elementi presenti nello stream. I due operatori richiedono in input un oggetto `java.util.Comparator` e restituiscono un oggetto `java.util.Optional` che è un oggetto contenitore che può o meno contenere un valore nullo.
- › **Count:** restituisce il numero di elementi nello stream dopo che è stato filtrato



## Esempi Stream (1) - Filtering

---

- › Restituire true se esiste un numero maggiore di 0

```
List<Integer> numbers = Arrays.asList(-2, 0, 10, 15, 3, -6);  
return numbers.stream().anyMatch(num -> num > 0);
```

- › Data una lista di Integer vogliamo avere solo quelli non nulli, maggiori di 0 e pari

```
numbers.stream().filter(Objects::nonNull)  
    .filter(num -> num > 0)  
    .filter(num -> num % 2 == 0);
```



## Esempi Stream (2) - Map

- › Datp uno Stream di Integer dove ogni elemento rappresenta il numero del mese, il metodo map eseguirà la conversione da numero a parola.

```
Stream<Integer> mesiIntStream =  
    Arrays.asList(0, 1, 2,...,11).stream();  
    //Stream.iterate(0, n -> n + 1).limit(12);  
Stream<String> mesiStringStream =  
    mesiIntStream.map(numb -> new  
        DateFormatSymbols().getMonths()[numb]);
```

DateFormatSymbols

[https://download.java.net/java/early\\_access/panama/docs/api/java.base/java/text/DateFormatSymbols.html](https://download.java.net/java/early_access/panama/docs/api/java.base/java/text/DateFormatSymbols.html)



## Esempi Stream (3) - Collecting

---

```
Stream<String> stream = Stream.of("bianco", "rosso", "giallo", "blu", "verde");
```

### › Liste e Set

```
List<String> listColori = stream.filter(colore ->  
    colore.contains("b")).collect(Collectors.toList());
```

```
Set<String> setColori = stream.filter(colore ->  
    colore.contains("b")).collect(Collectors.toSet());
```

### › String

```
String colori = stream.filter(colore ->  
    colore.contains("b"))  
    .collect(Collectors.joining(", "));
```