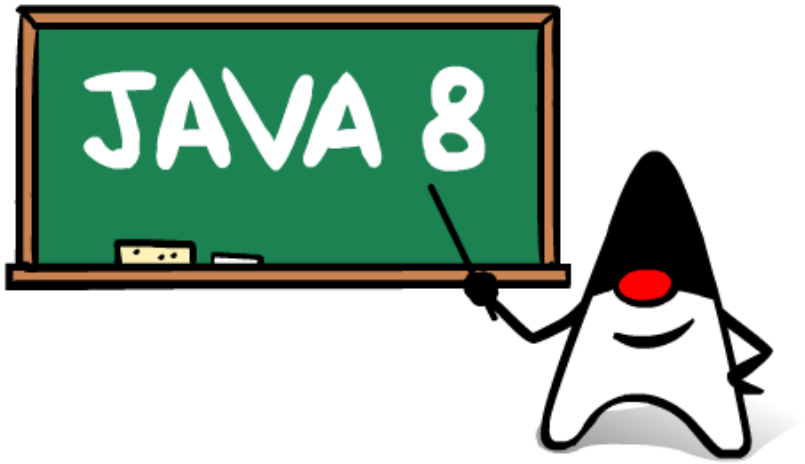
Metodologie di Programmazione

Lezione 32: Espressioni lambda

Lezione 32: Sommario

- Metodi di default
- Interfacce funzionali
- Espressioni lambda

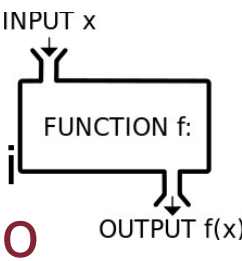


Programmazione funzionale in Java

- Java era finora un linguaggio incentrato sugli oggetti (orientato agli oggetti)
- Tuttavia, al giorno d'oggi nessun linguaggio può fare a meno di riferirsi a funzioni (come oggetti)
- Java 8 introduce importanti concetti di programmazione funzionale che permettono di definire funzioni e riferirsi ad esse come fossero implementazioni di interfacce

Che cos'è la programmazione funzionale

- Un paradigma di programmazione che tratta la computazione come la valutazione di funzioni matematiche
  - Evita la memorizzazione di uno stato e di dati mutabili
- E' un paradigma di programmazione dichiarativo
  - Esprime la logica della computazione (es. mediante espressioni) senza descrivere il controllo del flusso
  - **Che cosa**, non **come**!
- Si basa su **funzioni** che calcolano un risultato a partire da un certo input senza utilizzare uno stato del programma



Metodi di default per le interfacce

- E' possibile aggiungere metodi di default alle interfacce
- Utilizzando la parola chiave **default**

```
public interface Formula
{
    double calculate(int a);
    default double sqrt(int a) { return Math.sqrt(a); }
}
```

- Le classi concrete devono implementare solo i metodi astratti dell'interfaccia
- Utile per raggruppare metodi di utilità all'interno delle interfacce

Esempio (Java 7): una classe anonima che implementa l'interfaccia

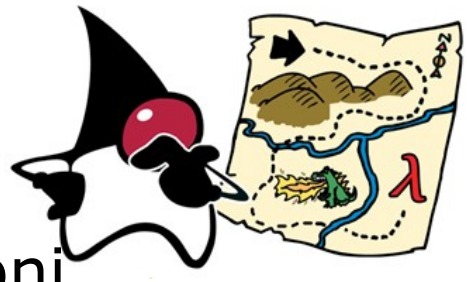

```
Formula formula = new Formula()
{
        @Override
        public double calculate(int a)
        {
                return sqrt(a * 100);
        }
};

formula.calculate(100); // 100.0
formula.sqrt(16); // 4.0
```

Interfacce funzionali

- E' disponibile una nuova annotazione @FunctionalInterface
- L'annotazione garantisce che l'interfaccia sia dotata esattamente di un solo metodo astratto
- Ad esempio:

```
@FunctionalInterface
public interface Runnable
{
   void run();
}
```

Espressioni lambda



- In Java 8 è possibile specificare funzioni utilizzando una notazione molto compatta, le espressioni lambda:

```
() -> { System.out.println("hello, lambda!"); }
```

- Tali espressioni creano oggetti anonimi assegnabili a riferimenti a interfacce funzionali compatibili con l'intestazione (input/output) della funzione creata

```
Runnable r = () -> { System.out.println("hello,
    lambda!"); };
r.run(); // stampa "hello, lambda!"
```

Sintassi delle espressioni lambda

- Un'espressione lambda è definita come segue:

(tipo1 nome_param1, …, tipon nome_paramn) -> { codice della funzione }

- Il tipo dei parametri in input è opzionale, perché desunto dal contesto (a quale interfaccia funzionale facciamo riferimento?)
- Le parentesi tonde sono opzionali se in input abbiamo un solo parametro
- Le parentesi graffe intorno al codice della funzione sono opzionali se è costituito da una sola riga
- Non è necessario return, se il codice è dato dall'espressione di ritorno

Espressioni lambda equivalenti

- (int a, int b) -> { return a+b; }
- (a, b) -> { return a+b; }
- (a, b) -> return a+b;
- (a, b) -> a+b;

- (String s) -> { return s.replace(' ', '_'); }
- (s) -> { return s.replace(' ', '_'); }
- s -> { return s.replace(' ', '_'); }
- s -> return s.replace(' ', '_');
- s -> s.replace(' ', '_');

Esempi di espressioni lambda

- (int a, int b) -> a+b
- a -> a*a
- () -> System.out.println("Hello, world!");
- s -> System.out.println(s);
- () -> 42;
- () -> 3.1428;
- (String s) -> s.length();
- s -> s.length();

Consideriamo nuovamente il nostro esempio


```
public interface Formula
{
        double calculate(int a);
        default double sqrt(int a) { return Math.sqrt(a); }
}
```

Esempio di implementazione di Formula prima e dopo Java 8

In Java 7:

```
Formula formula = new Formula() {
        @Override
        public double calculate(int a)
        {
        return sqrt(a * 100);
        }
    };
```

In Java 8:

```
Formula formula = a -> Math.sqrt(a*100);
Formula formula2 = a -> a*a;
Formula formula3 = a -> a-1;
```

Esempio: conversione da un tipo F a un tipo T


```
@FunctionalInterface
public interface Converter<F, T>
{
        T convert(F from);
}
```

```
Converter<String, Integer> converter = from -> Integer.valueOf(from);
Integer converted = converter.convert("123");
Converter<String, MyString> stringConverter = a -> new MyString(a);
MyString myString = stringConverter.convert("123");
```

Single Abstract Method (SAM) type

- Le interfacce funzionali sono di tipo SAM
- A ogni metodo che accetta un'interfaccia SAM, si può passare un'espressione lambda compatibile con l'unico metodo dell'interfaccia SAM
- Analogamente per un riferimento a un'interfaccia SAM

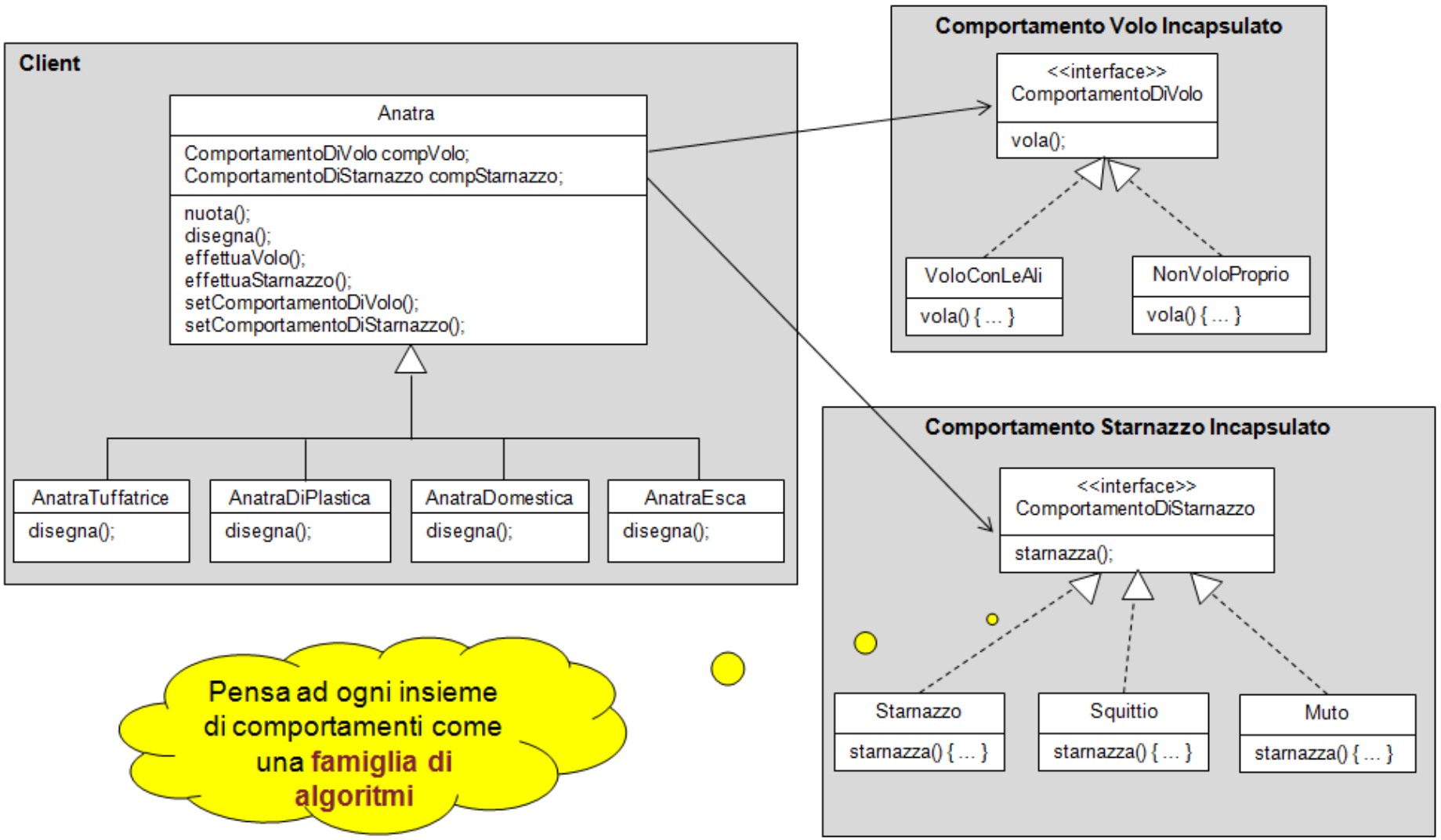
Riferimenti a metodi esistenti

- E' possibile passare riferimenti a metodi esistenti
- Utilizzando la sintassi:
  - Classe::metodoStatico
  - riferimentoOggetto::metodo
  - Classe::metodoNonStatico (vedi più avanti)

- Esempio:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
```

Ricordate lo Strategy pattern?



Strategy pattern in Java 8!

- Le varie implementazioni di un metodo possono essere fornite direttamente in input come espressioni lambda!
- Es.

```
ComportamentoDiStarnazzo st;
…
st = Starnazzo::starnazza;
st = Squittio::starnazza;
st = Muto::starnazza;
```

Riferirsi a metodi d'istanza con il nome della classe vs. usando un riferimento a un oggetto

- Che differenza c'è tra:
  - riferimentoOggetto::metodo e
  - nomeClasse::metodo?
- Nel secondo caso, non stiamo specificando su quale oggetto applicare il metodo
  - Ma il metodo è d'istanza, quindi utilizza membri d'istanza (campi, metodi, ecc.)
- Nel secondo caso, ci si riferisce al metodo con un primo parametro aggiuntivo: un riferimento alla classe cui appartiene il metodo

Esempio di riferimento a metodi d'istanza mediante classe

- Considerate Arrays.sort(T[] a, Comparator<? super T> c)
- E Comparator:

```
public interface Comparator<T>
{
    int compare(T o1, T o2);
    boolean equals(Object o);
}
```

- Posso scrivere:

```
Arrays.sort(new String[] { "a", "c", "b" }, String::compareTo);
```

Esempio di riferimento a metodi d'istanza mediante classe


```
public interface StringProcessor
{
   String process(String s);
}

StringProcessor f = String::toLowerCase;
System.out.println(f.process("BELLA"));   // stampa "bella"
String s = "bella";
StringProcessor g = s::concat;   // stesso di: "bella"::concat
System.out.println(g.process(" zi'!")); // stampa "bella zi'!"
// metodo statico String valueOf(Object o)
StringProcessor h = String::valueOf;
System.out.println(h.process("bella"));    // stampa "bella"
```

Riferimento a costruttori esistenti

- E' possibile passare riferimenti a costruttori esistenti
- Utilizzando la sintassi:
  - Classe::new
- Perfetto per il (simple) factory pattern!

Esempio (1)


```
public class Person
{
        private String firstName;
        private String lastName;
        public Person(String firstName, String lastName)
        {
        this.firstName = firstName; this.lastName = lastName;
        }
}

@FunctionalInterface
public interface PersonFactory<P extends Person>
{
        P create(String firstName, String lastName);
}
```

Esempio (2)


```
@FunctionalInterface
public interface PersonFactory<P extends Person>
{
        P create(String firstName, String lastName);
}

PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

- Invece di creare la factory manualmente, utilizziamo i riferimenti ai metodi new compatibili con l'interfaccia di factory e in qualsiasi momento possiamo costruire nuovi oggetti nel modo desiderato!

Visibilità dalle espressioni lambda

- Accesso alla variabili esterne da un'espressione lambda molto simile a quello di un oggetto anonimo
- Si possono accedere:
  - Campi d'istanza e variabili statiche
  - Variabili final del metodo esterno

- Esempio:

```
final int num = 1;
Converter<Integer, String> stringConverter =
                    from -> String.valueOf(from + num);
stringConverter.convert(2); // "3"
```

Visibilità dalle espressioni lambda

- Accesso alla variabili esterne da un'espressione lambda molto simile a quello di un oggetto anonimo
- Si possono accedere:
  - Campi d'istanza e variabili statiche
  - Variabili final del metodo esterno
  - Variabili del metodo esterno **implicitamente final**
- Esempio:

```
int num = 1; // non viene modificata all'interno del metodo
Converter<Integer, String> stringConverter =
                    (from) -> String.valueOf(from + num);
stringConverter.convert(2); // "3"
```

Visibilità dalle espressioni lambda

- Accesso alla variabili esterne da un'espressione lambda molto simile a quello di un oggetto anonimo
- Si possono accedere:
  - Campi d'istanza e variabili statiche
  - Variabili final del metodo esterno
  - Variabili del metodo esterno **implicitamente final**
- Tuttavia non è possibile accedere ai metodi di default dall'interno di un'espressione lambda
- Ad esempio, essendo sqrt di default, non si può scrivere:

```
Formula formula = a -> sqrt(a * 100);
```

NO!!!

Ordinamento inverso: da Java 7 a Java 8

In Java 7:

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
Collections.sort(names, new Comparator<String>()
{
    @Override
    public int compare(String a, String b) { return b.compareTo(a); }
});
```

In Java 8:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Ordinamento per lunghezza: da Java 7 a Java 8

In Java 7:

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
Collections.sort(names, new Comparator<String>()
{
    @Override
    public int compare(String a, String b)
    {
        return Integer.compare(a.length(), b.length());
    }
});
```

In Java 8:

```
Collections.sort(names, (a, b) -> a.length()-b.length());
                    // oppure: (a, b) -> Integer.compare(a.length(), b.length());
```

Esercizio: array con ordinamento dinamico in Java 8

- Svolgere nuovamente l'esercizio sull'array con ordinamento dinamico utilizzando il paradigma funzionale per implementare lo strategy pattern
  - Utilizzare anche il metodo di default reversed dell'interfaccia Comparator (disponibile in Java 8)

Lancio di un thread da Java 7 a Java 8

In Java 7:

```
new Thread(new Runnable()
   {
       @Override
       public void run() { System.out.println("Hello from thread");
    }).start();
```

In Java 8:

```
new Thread(() -> System.out.println("Hello from thread")).start();
```

Ascolto degli eventi da Java 7 a Java 8


```
JButton button = new JButton("Test Button");

// In Java 7:
button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae)
        {
           System.out.println("Click Detected by Anonymous Class");
        }});

// In Java 8:
button.addActionListener(e -> System.out.println("Click Detected by Lambda Listener"));

// Swing stuff
JFrame frame = new JFrame("Listener Test");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.add(button, BorderLayout.CENTER);
frame.pack();
frame.setVisible(true);
```

Metodi di default nell'interfaccia Comparator


```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);
Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");
comparator.compare(p1, p2); // > 0
```

- Confronto inverso:

```
comparator.reversed().compare(p1, p2); // < 0
```

- Confronto per una data chiave specificata:

```
comparator = Comparator.comparing(p -> p.getFirstName());
```

- Confronto per criteri multipli a cascata:

```
comparator = Comparator.comparing(p -> p.getFirstName())
        .thenComparing(p -> p.getLastName());
```

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione a valori booleani a un solo argomento generico T

```
Predicate<String> predicate = s -> s.length() > 0;
    predicate.test("foo"); // true
predicate.test(""); // false
```

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione a valori booleani a un solo argomento generico T

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
default **Predicate**<T>	**and**(**Predicate**<? super T> other) Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> **Predicate**<T>	**isEqual**(**Object** targetRef) Returns a predicate that tests if two arguments are equal according to **Objects.equals(Object, Object)**.
default **Predicate**<T>	**negate**() Returns a predicate that represents the logical negation of this predicate.
default **Predicate**<T>	**or**(**Predicate**<? super T> other) Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	**test**(T t) Evaluates this predicate on the given argument.

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione a valori booleani a un solo argomento generico T

```
Predicate<String> predicate = s -> s.length() > 0;
    predicate.test("foo"); // true
predicate.negate().test("foo"); // false
Predicate<String> predicate2 = s -> s.startsWith("f");
predicate.and(predicate2).test("foo"); // true
Predicate<Boolean> nonNull = Objects::nonNull;
    Predicate<Boolean> isNull = Objects::isNull;
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione booleana a un solo argomento generico T
- Function<T,R>: funzione a un argomento di tipo T e un tipo di ritorno R entrambi generici

```
Function<String, Integer> toInteger = Integer::valueOf;
Integer k = toInteger.apply("123");
Function<Integer, Integer> square = k -> k*k;
Integer sqr = square.apply(5); // 25
```

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione booleana a un solo argomento generico T
- Function<T,R>: funzione a un argomento di tipo T e un tipo di ritorno R entrambi generici

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`default <V> Function<T,V>`	**`andThen(Function<? super R,? extends V> after)`** Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.
`R`	**`apply(T t)`** Applies this function to the given argument.
`default <V> Function<V,R>`	**`compose(Function<? super V,? extends T> before)`** Returns a composed function that first applies the `before` function to its input, and then applies this function to the result.
`static <T> Function<T,T>`	**`identity()`** Returns a function that always returns its input argument.

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione booleana a un solo argomento generico T
- Function<T,R>: funzione a un argomento di tipo T e un tipo di ritorno R entrambi generici

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<Integer, String> toString = String::valueOf;
Function<String, String> backToString =
        toInteger.andThen(toString);
backToString.apply("123"); // "123"
```

Esempio


```
class Person
{
        private String firstName;
        private String lastName;

        public Person() {}
        public Person(String firstName, String lastName)
        {
                this.firstName = firstName;
                this.lastName = lastName;
        }
}
```

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione booleana a un solo argomento generico T
- Function<T,R>: funzione a un argomento di tipo T e un tipo di ritorno R entrambi generici
- Supplier<T>: funzione senza argomenti in input e un tipo di ritorno T generico

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get(); // new Person()
```

Interfacce funzionali built-in (java.util.function.*)

- Predicate<T>: funzione booleana a un solo argomento generico T
- Function<T,R>: funzione a un argomento di tipo T e un tipo di ritorno R entrambi generici
- Supplier<T>: funzione senza argomenti in input e un tipo di ritorno T generico
- Consumer<T>: funzione con un argomento di tipo generico T e nessun tipo di ritorno

```
Consumer<Person> greeter = p ->
    System.out.println("Hello, " + p.firstName);
// oppure greeter = System.out::println;
greeter.accept(new Person("Luke", "Skywalker"));
```

For each su java.util.Collection

- Le collection sono ora dotate di un metodo forEach che prende in input un'interfaccia Consumer<? super T> dove T è il tipo generico della collection
- Ad esempio:

```
Collection<String> c = Arrays.asList("aa", "bb", "cc");

// In Java 7:
for (String s : c) System.out.println(s);

// In Java 8:
c.forEach(s -> System.out.println(s));
// persino meglio:
c.forEach(System.out::println);
```

Stanco delle NullPointerException?

- java.util.Optional è un contenitore di un riferimento che potrebbe essere o non essere null
- Un metodo può restituire un Optional invece di restituire un riferimento potenzialmente null

```
Optional<String> optional = Optional.ofNullable("bam");
optional.isPresent(); // true
optional.get(); // "bam"
optional.orElse("fallback"); // "bam"
optional.ifPresent(s -> System.out.println(s.charAt(0))); // "b"
```

Un nuovo meccanismo per le sequenze di elementi: gli Stream

- Una nuova interfaccia java.util.stream.Stream
- Rappresenta una sequenza di elementi su cui possono essere effettuate una o più operazioni
- Supporta operazioni sequenziali e paralelle
- Le operazioni possono essere intermedie o terminali
  - **Intermedie:** restituiscono un altro stream su cui continuare a lavorare
  - **Terminali:** restituiscono il tipo atteso
- Uno Stream viene creato a partire da una sorgente, ad esempio una java.util.Collection

Metodi di Stream: filter, forEach

- filter è un metodo di Stream che accetta un predicato (Predicate) per filtrare gli elementi dello stream
  - Operazione intermedia che restituisce lo stream filtrato
- forEach prende in input un Consumer e lo applica a ogni elemento dello stream
  - Operazione terminale

Esempio di Stream (con filter e forEach)

- Filtra gli elementi di una lista per iniziale e stampa ciascun elemento rimanente:

```
List<String> l = Arrays.asList("da", "ab", "ac", "bb");
l.stream()
  .filter(s -> s.startsWith("a"))
  .forEach(System.out::println);
```

Altro esempio di Stream (con filter e forEach)

- Filtra gli elementi di una lista per iniziale e lunghezza della stringa e stampa ciascun elemento rimanente:

```
Predicate<String> startsWithJ = s -> s.startsWith("J");
Predicate<String> fourLetterLong = s -> s.length() == 4;

List<String> l = Arrays.asList("Java", "Scala", "Lisp");
l.stream()
  .filter(startsWithJ.and(fourLetterLong)
  .forEach(s -> System.out.println("Inizia con J ed e' lungo 4
    caratteri: "+s);
```

Metodi di Stream: sorted

- sorted è un'operazione intermedia sugli stream che restituisce una vista ordinata dello stream **senza modificare la collezione sottostante**
- Esempio:

```
List<String> l = Arrays.asList("da", "ac", "ab", "bb");
l.stream()
  .sorted()
  .filter(s -> s.startsWith("a"))
  .forEach(System.out::println);
```

- Stampa:

```
ab
ac
```

Metodi di Stream: map

- map è un'operazione intermedia sugli stream che converte ciascun elemento in un altro oggetto attraverso la funzione (Function) passata in input
- Esempio:

```
l.stream().map(String::toUpperCase).sorted((a, b) ->
    b.compareTo(a)) .forEach(System.out::println);
```

- Stampa:

```
DA
BB
AC
AB
```

Stream.map: esempio

- Si vuole scrivere un metodo che aggiunga l'IVA a ciascun prezzo:

```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);

// In Java 7:
for (int p : ivaEsclusa)
{
    double pIvaInclusa = p*1.22;
    System.out.println(pIvaInclusa);
}
// In Java 8:
ivaEsclusa.stream().map(p ->
    p*1.22).forEach(System.out::println);
```

Metodi di stream: collect

- collect è un'operazione terminale che permette di raccogliere gli elementi dello stream in un qualche oggetto (ad es. una collection)
- Ad esempio, per ottenere la lista dei prezzi ivati:

```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);
// In Java 7:
List<Double> l = new ArrayList<>();
for (int p : ivaEsclusa) l.add(p*1.22);
// In Java 8:
List<Double> l = ivaEsclusa.stream().map(p -> p*1.22)
                                    .collect(Collectors.toList());
```

Esempio: creare una stringa che concateni stringhe in una lista, maiuscole e separate da ,


```
List<String> l = Arrays.asList("RoMa", "milano",
"Torino");
String s = "";

// in Java 7:
for (String e : l)
s += e.toUpperCase()+", ";
s = s.substring(0, s.length()-2);

// in Java 8:
s = l.stream().map(e -> e.toUpperCase())
                    .collect(Collectors.joining(", "));
```

Esempio: creare una sottolista di valori distinti


```
List<Integer> l = Arrays.asList(3, 4, 5, 3, 4, 1);
List<Integer> distinti = l.stream().map(x -> x*x)

.distinct().collect(Collectors.toList());
```

Metodi di Stream: reduce

- reduce è un'operazione terminale che effettua una riduzione sugli elementi dello stream utilizzando la funzione data in input
- Esempio:

```
Optional<String> reduced = l.stream().sorted()
    .reduce((s1, s2) -> s1 + "#" + s2);
reduced.ifPresent(System.out::println); // "ab#ac#bb#da"
```

Esempio: calcolo della somma dei valori iva inclusa


```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);

// In Java 7:
double totIvaInclusa = 0.0;
for (int p : ivaEsclusa)
{
    double pIvaInclusa = p*1.22;
    totIvaInclusa += pIvaInclusa;
}
// In Java 8:
ivaEsclusa.stream().map(p -> p*1.22).reduce((sum, p) ->
sum+p).get();
```

- reduce restituisce un Optional (per ottenere il valore, si utilizza il metodo get())

Metodi di Stream: count

- count è un'operazione terminale che restituisce il numero long di elementi nello stream
- Esempio:

```
long startsWithA = l.stream().filter((s) -> s.startsWith("a")) .count();
System.out.println(startsWithA); // 2
```

Stream paralleli

- Le operazioni su stream sequenziali sono effettuate in un singolo thread
- Le operazioni su stream paralleli, invece, sono effettuate concorrentemente su thread multipli

Perché creare uno stream parallelo?

- Esempio:

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++)
{
   UUID uuid = UUID.randomUUID();
   values.add(uuid.toString());
}
```

Perché creare uno stream parallelo?

- Con **l'ordinamento sequenziale**:

```
long t0 = System.nanoTime();
long count = values.stream().sorted().count();
System.out.println(count);
long t1 = System.nanoTime();
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("ordinamento sequenziale: %d
    ms", millis));
// ordinamento sequenziale: 899 ms
```

Perché creare uno stream parallelo?

- Con **l'ordinamento parallelo**:

```
long t0 = System.nanoTime();
long count = values.parallelStream().sorted().count();
System.out.println(count);
long t1 = System.nanoTime();
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("ordinamento parallelo: %d
    ms", millis));
// ordinamento parallelo: 472 ms
```

Le mappe in Java 8

- Le mappe non supportano gli stream
- Ma forniscono numerose operazioni aggiuntive in Java 8

```
Map<Integer, String> map = new HashMap<>();
for (int i = 0; i < 10; i++) map.putIfAbsent(i, "val" + i);
map.forEach((id, val) -> System.out.println(val));
```

Le mappe in Java 8

- Se l'elemento 3 è presente, modifica il valore associatogli utilizzando la BiFunction in input come secondo parametro:

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3); // val33
map.computeIfPresent(9, (num, val) -> null);
    map.containsKey(9); // false
map.computeIfAbsent(23, num -> "val" + num);
    map.containsKey(23); // true
map.computeIfAbsent(3, num -> "bam");
map.get(3); // val33
```

Le mappe in Java 8


```
map.remove(3, "val3");
map.get(3); // val33
map.remove(3, "val33");
map.get(3); // null
map.getOrDefault(42, "not found"); // not found

map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9); // val9
map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9); // val9concat
```

Differenze tra classi anonime ed espressioni lambda

- La parola chiave this:
  - **Classi anonime**: si riferisce all'oggetto anonimo
  - **Espressioni lambda**: si riferisce all'oggetto della classe che le racchiude
- La compilazione è differente:
  - **Classi anonime**: compilate come classi interne
  - **Espressioni lambda**: compilate come metodi privati invocati dinamicamente