**Equals**

TIPI con EQUALS già implementato di default:

Boolean,

Byte,

Character,

Double,

Float,

Integer,

Long,

Short,

String,

Enum.

---

```
public final boolean equals(Object other) {
      if(other instance of TipoDiThis {
            TipoDiThis  otherAsTipoDiThis = (TipoDiThis) other;
            return campoX ==  otherAsTipoDiThis.campoX &&
                        campoY ==  otherAsTipoDiThis.campoY &&
                        campoZ ==  otherAsTipoDiThis.campoZ;
      }
      else
            return false;
}
```

---

NOTE:

Se uno dei campi è del tipo di quelli che hanno equals di default fai:

```
campoW.equals(otherAsTipoDiThis.campoW)
```

**CompareTo**

TIPI con CompareTo già implementato di default:

Byte,

Character,

Double,

Float,

Integer,

Long,

Short,

String,

Enum.

implements Comparable<T>

```
public final int compareTo(T other) {
      int diff = FZ(other);
      if (diff != 0)
            return diff;
      diff = name.compareTo(other.name);
      if (diff != 0)
            return diff;
      return Double.compare(price, other.price);
}
```

NOTE:

posso usare una funzione per diff.

Se il CompareTo è già implementato di default allora uso quello per diff.

Posso usare anche per tipi speciali Double.compare(campoW, other.campoW ), lo stesso per Long.

**HashCode**

TIPI con HashCode già implementato di default:

Boolean,

Byte,

Character,

Double,

Float,

Integer,

Long,

Short,

String,

Enum

---

```
public int hashCode() {
return year ^ month ^ day ^ moment.hashCode();
}
```

---

NOTE:

- se è gia implementato per default campoW.hashCode();
- se è un numero ma non un intero fai un cast ad int.

Esempio:

```
public final int hashCode() { return name.hashCode() ^ (int) price; }
```

In ogni caso su tutti i campi dell'oggetto usi l'operatore XOR(^).

HashCode ed Equals devono essere corretti e coerenti:

Se due oggetti sono considerati uguali secondo la logica dell'applicazione, il metodo `equals()` dovrebbe restituire `true` quando confronta questi due oggetti.

Inoltre, se `equals()` restituisce `true` per due oggetti, allora il metodo `hashCode()` dovrebbe restituire lo stesso valore di hash per entrambi gli oggetti.

**ECCEZIONI**

CHECKED vuol dire controllate → sono da dichiarare (throws …)R/W file:

IOException, FileNotFoundException

UNCHECKED → NON sono da dichiarare

RuntimeException:

IllegalArgumentException, NullPointerException, ArithmeticException

PER DEFINIRICI UNA NOSTRA ECCEZIONE:

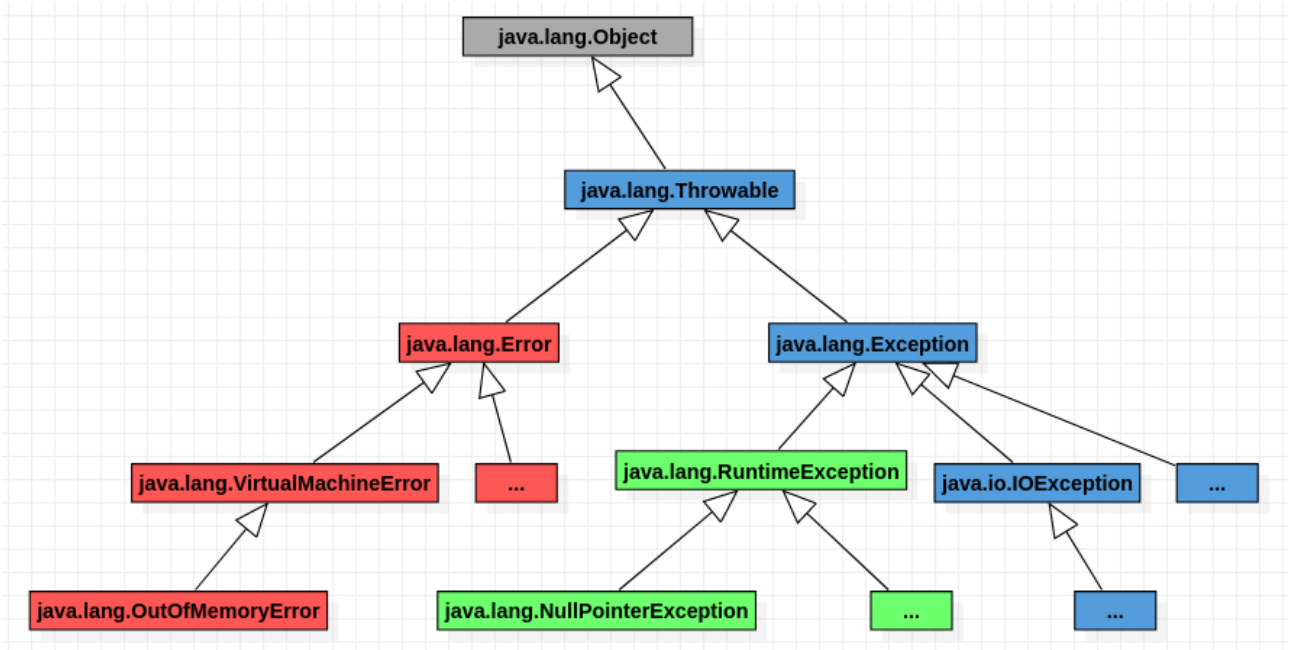
CLASSE extends RuntimeException{

COSTRUTTORE(String message){

Super(message)

}

}




ESEMPIO D'USO ECCEZIONE DEFINITA DAL PROGRAMMATORE

```
public class IllegalDateException extends RuntimeException {
    public IllegalDateException(String message) {
       super(message);
    }

}
```

ESEMPIO D'USO ECCEZIONE UNCHECKED(salvo casi particolari non serve dichiararle)

```
if (key == null)
      throw new NullPointerException("null keys are not allowed");
```

ESEMPIO D'USO ECCEZIONE CHECKED(o le gestisco(circondo con try e catch) o le dichiaro = le lascio propagare al metodo che ha chiamato il metodo che l'ha generata)

```
      public Polish(String fileName) throws FileNotFoundException {

            this.writer = new PrintWriter(fileName);

      }
```

Gestione eccezioni:

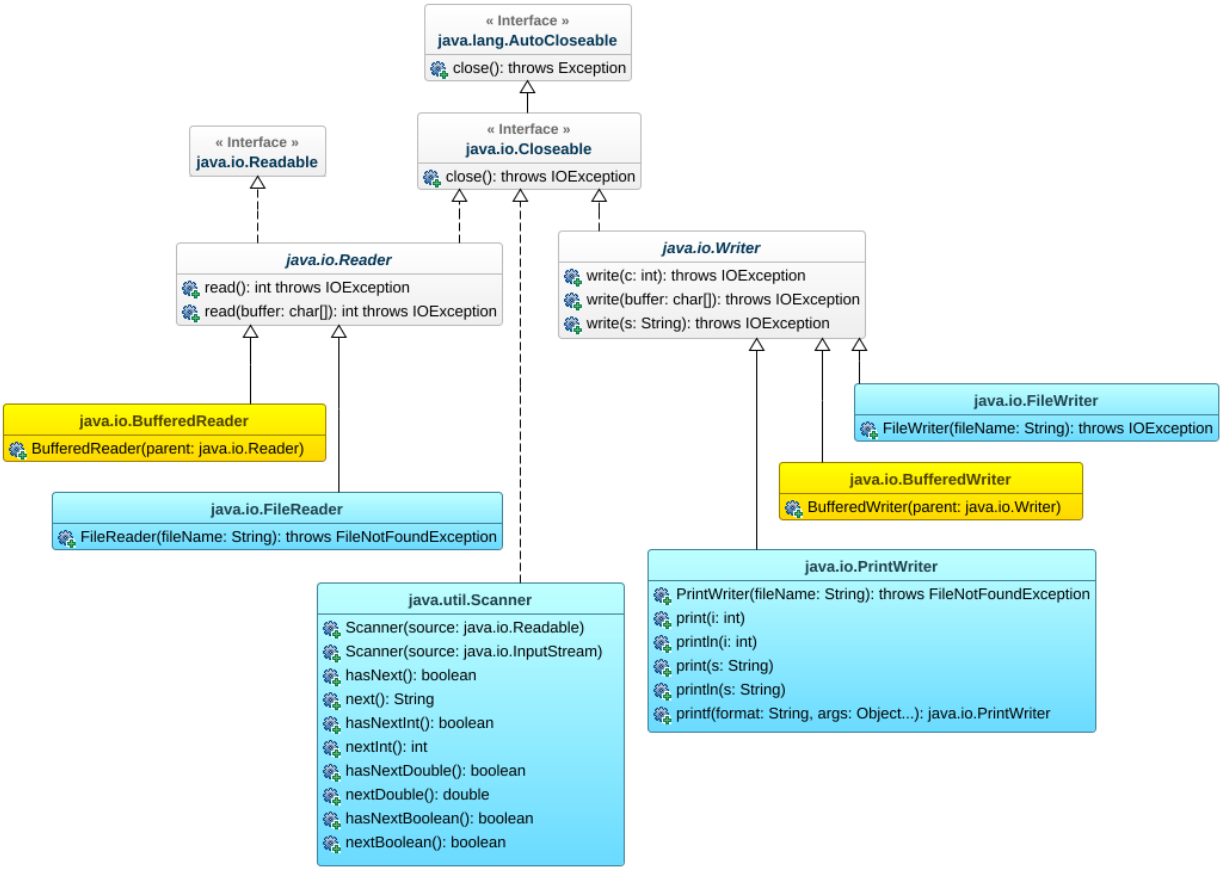
TRY/CATCH/FINALLY

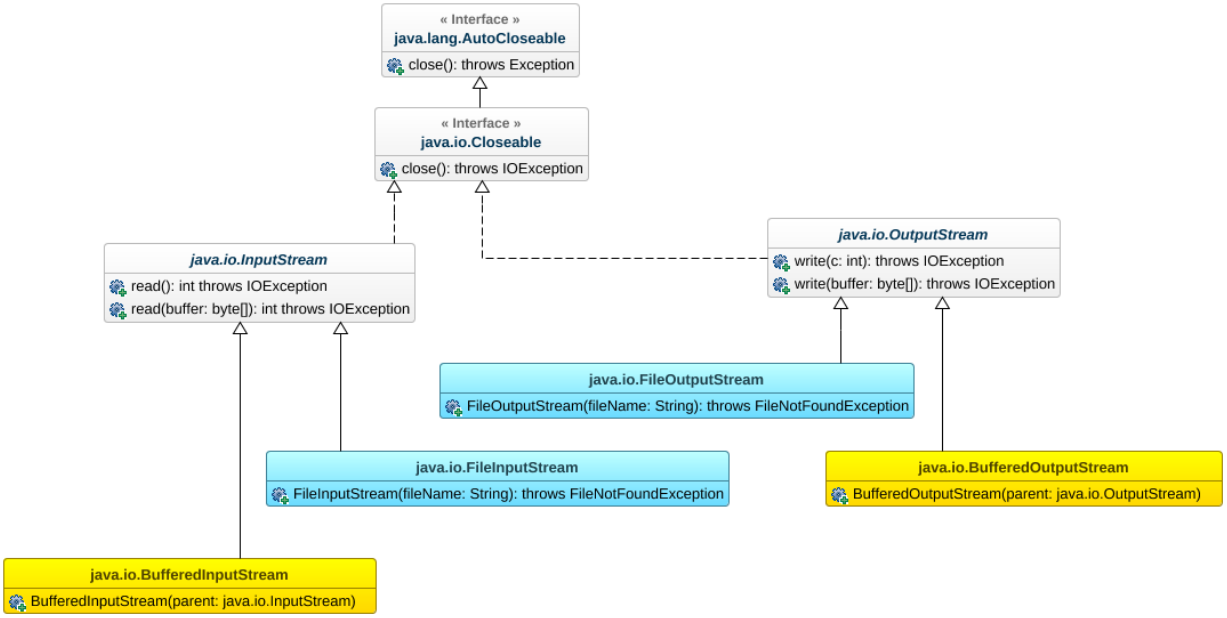
```
try {
      ...codice che potrebbe lanciare un'eccezione...
} catch(Exception E) {
      ...stampa di un messaggio...
} finally {
      ...codice che viene eseguito sempre...(usato per chiudere risorse)
}
```

**R/W FILE**

## TRY WITH RESOURCES

```
public static void main(String[] args) {
    try (Reader reader = new FileReader("commedia.txt")) {
       int c;
       while ((c = reader.read()) != -1)
          System.out.print((char) c);
    }
    catch (FileNotFoundException e) {
       System.out.println("File non trovato!");
    }
    catch (IOException e) {
       System.out.println("Errore di I/O");
    }
}
```

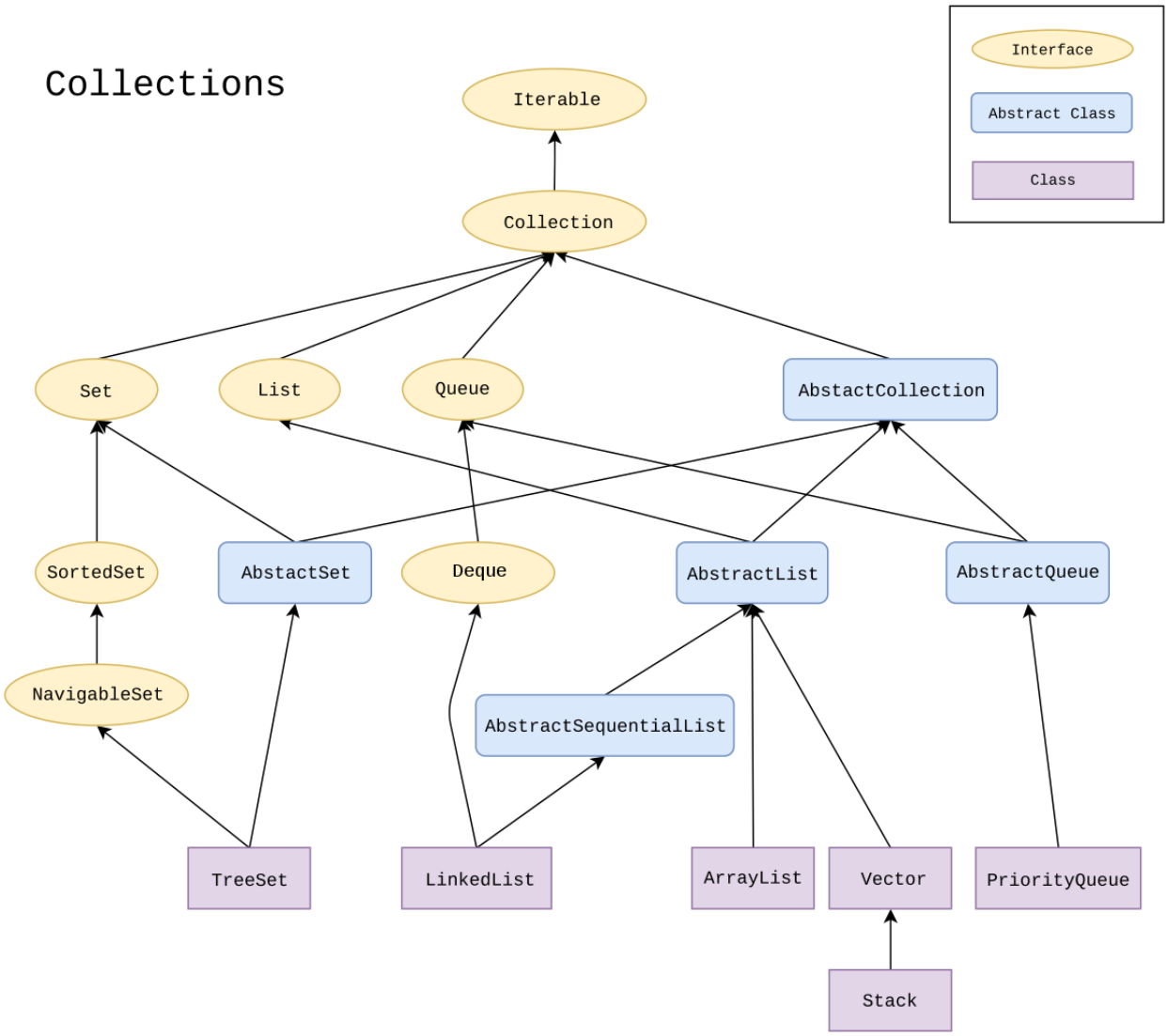





```
try (Reader reader = new BufferedReader(new FileReader("commedia.txt"));
     Writer writer = new BufferedWriter(new FileWriter("copia.txt"))) {

    int c;
    while ((c = reader.read()) != -1)
       writer.write(c);
}
catch (FileNotFoundException e) {
    System.out.println("Il file non esiste!");
}
catch (IOException e) {
    System.out.println("Errore di I/O");
}
```

# COLLEZIONI


Collections
Interface
Abstract Class
Class
Iterable
Collection
Set
List
Queue
AbstactCollection
SortedSet
AbstactSet
Deque
AbstractList
AbstractQueue
NavigableSet
AbstractSequentialList
TreeSet
LinkedList
ArrayList
Vector
PriorityQueue
Stack

# LISTE

Caratteristiche: Sequenza di elementi → sono ammessi doppi e non ho un ordine

## ArrayList

Quando usarlo, per cosa viene usato?

- Se l'acceso casuale è frequente
- Inserimento/rimozione di elementi in posizioni specifiche non è frequente
- Devo aggiungere in coda

*ESEMPIO D'USO:*

```
List<String> result = new ArrayList<>();

for (MultiWordIdentifier id: ids)

     for (String s: id.words)

          result.add(s);

return result;
```

NOTE/CONDIZIONI/VINCOLI

Trucchetto perché il varargs non ammetta lo zero : `add(Coin first, Coin … all)`

## LinkedList

Quando usarlo, per cosa viene usato?

- Accesso casuale non è frequente
- Inserimento/rimozione in posizioni specifiche è frequente
- Devo aggiungere in testa

*ESEMPIO D'USO:*

```
        LinkedList<String> FruitsList = new LinkedList<>();


        // Aggiunta di elementi alla LinkedList

        linkedList.add("Apple");

        linkedList.add("Banana");

        linkedList.add("Orange");


        // Stampa della LinkedList

        System.out.println("LinkedList: " + FruitsList);
```

# INSIEMI

Caratteristiche: Raccolta di elementi → non sono ammessi doppi, ho un ordine nel TreeSet mentre invece nell'HashSet non ho un ordine

## TreeSet

Quando usarlo, per cosa viene usato?

- Ho bisogno di una raccolta ordinata di elementi senza duplicati

*ESEMPIO D'USO:*

```
SortedSet<Slot> slots = new TreeSet<>();

for (Set<Slot> set: availabilities.values())

     slots.addAll(set);
```

*Metodi tipici:* first→ritorna elemento più piccolo, last→ritorna elemento più grande

NOTE/CONDIZIONI/VINCOLI

- I tipi degli elementi devono essere Comparable
- I duplicati vengono rilevati con equals → dove non è gia definito va ridefinito
- Dove viene ridefinito equals va ridefinito anche l'hashcode ( va garantita la coerenza )

## HashSet

Quando usarlo, per cosa viene usato?

- Ho bisogno di una raccolta non ordinata di elementi e senza duplicati

*ESEMPIO D'USO:*

```
Set<Tessera<T>> insieme = new HashSet<>();

while (insieme.size() < width * height - 1) {

     Tessera<T> tessera = fattoria.get();

     if (insieme.add(tessera))

         // se e' stato aggiunto all'insieme, vuol dire che
         // non e' ripetuto e lo aggiungiamo anche alla lista
          this.tessere.add(tessera);

}
```

NOTE/CONDIZIONI/VINCOLI

- Equals simmetrico e transitivo
- Equals e hashcode compatibili
- Hashcode non banale

# CODE

Caratteristiche:

Code di elementi → sono ammessi doppi, ho un ordine, aggiungo da una parte ed estraggo dalla parte opposta

## PriorityQueue

Quando usarlo, per cosa viene usato?

- Ho bisogno di una coda con ordinamento basato su priorità

*ESEMPIO D'USO:*

```
PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();

        // Aggiunta di elementi alla PriorityQueue
        priorityQueue.add(10);
        priorityQueue.add(5);

        // Stampa della PriorityQueue
        System.out.println("PriorityQueue: " + priorityQueue);

        // Ottenere e rimuovere l'elemento con la massima priorità
        int highestPriorityElement = priorityQueue.poll();
        System.out.println("Elemento con la massima priorità: " +
                                                highestPriorityElement);
```

NOTE/CONDIZIONI/VINCOLI

L'elemento ad alta priorità è il più piccolo, comunque sia la priorità dipende dal CompareTo che è ridefinibile

# MAPPE

Caratteristiche:

Mapping chiave<-->valore dove ad ogni chiave è associato un valore e le chiavi sono univoche.

HashMap non mantiene l'ordine delle chiavi.

TreeMap mantiene l'ordine delle chiavi.

LinkedHashMap → l'ordine è basato sull'ordine di inserimento

## HashMap

Quando usarlo, per cosa viene usato?

*ESEMPIO D'USO:*

```
private final Map<Partito, Integer> voti = new HashMap<>();

// registra un voto per il partito indicato

public final void vota(Partito partito) {

      voti.put(partito, 1 + votiPer(partito));

}
```

*Metodi tipici:*

put(K key, V value) → setta valore per chiave e ritorna il vecchio valore se era presenta altrimenti null

V get (Object key) →legge una chiave e ottiene il valore associato

putIfAbsent(K key, V value)→ rimpiazza la coppia chiave(k)-valore(value) solo se la chiave(key) non era già presente nel mappaggio e in quest caso ritorna null, mentre se invece la chiave era già presente nel mappaggio non viene eseguita alcuna sostituzione e si ritorna il valore associato a quella chiave nel mapping presente attuale.

Remove(K key)→rimuove il legame

Size → ritorna quantità di legami

Contains(K key)→ per sapere se una chiave è già stata legata

Contains(V value)→per sapere se un valore è già stato legato

Keyset() →ritorna insieme delle chiavi

Values() →ritorna insieme di valori

NOTE/CONDIZIONI/VINCOLI

- Le chiavi devono implementare hashcode e equals compatibili

# THREAD


→ quello che deve eseguire il thread

→ fa partire il run() in parallelo al codice dopo start()

→ resto in attesa fintantoché il thread non ha finito

→ interrompo il thread

→ aspetto una quantità definita di millisecondi

→ mi dice qual è il thread in esecuzione

A thread can be created by subclassing the java.lang.Thread class:

```
public class MyThread extends Thread {
	@Override
	public void run() { ... do something here ... }
}
...
new MyThread().start();
```

A thread can also be created by specifying the task in the constructor of anew java.lang.Thread:

```
public class MyRunnable implements Runnable {
	@Override
	public void run() { ... do something here ... }
}
...
new Thread(new MyRunnable()).start();
```

Per far si che più thread possano lavorare in parallelo devo assicurarmi che siano sincronizzati per evitare situazioni come deadlock o dati inconsistenti.

Per tale motivo occorre ricreare “semafori” in quelle che sono le sezioni critiche.

Un modo semplice per farlo è questo:

Creo un oggetto statico Object o ed inserisco le zone critiche dentro synchronized(o){…}:

```
public class RaceCondition extends Thread {
    private static long nice;
    private static long ugly;
    private static Object o = new Object();

    @Override
    public void run() {
       for (long count = 0; count < 100000L; count++) {
          synchronized(o) {
             nice++;
             ugly++;
          }
       }
     }

      //[…]

}
```

Vediamo un altro esempio di codice:

```
private class Worker extends Thread {
    public void run() {
        // body del thread Worker
    }
}
```

```
public class Catch22Parallel {
    private AtomicInteger candidate = new AtomicInteger(10);
    private AtomicInteger counter = new AtomicInteger(0);

    private Catch22Parallel() throws InterruptedException {
       Worker[] workers = new Worker[Runtime.getRuntime().availableProcessors()];
       for (int pos = 0; pos < workers.length; pos++)
          workers[pos] = new Worker();
       for (Worker worker: workers)
          worker.start();
       for (Worker worker: workers)
          worker.join();
    }
```

# LAMBDA

Le lambda expressions sono implementazioni anonime di interfacce funzionali.

Un' interfaccia funzionale è un' interfaccia che contiene al suo interno una sola dichiarazione di un unico metodo astratto.

Esempi di interfacce funzionali:

```
public interface Consumer<T> {
	void accept(T);
}

public interface UnaryOperator<T> {
	T apply(T t);
}
```

Interfacce funzionali più comuni per cui si usano le lambda con esempi:

| Interface | Args. | Returns | Example |
|---|---|---|---|
| Consumer<T> | T | void | s -> System.out.print(s) |
| Predicate<T> | T | boolean | s -> s.isEmpty() |
| Supplier<T> | none | T | () -> new String() |
| Function<T,U> | T | U | s -> new Integer(s) |

Riferimenti a metodi: sintassi più semplice delle lambda applicabile quando ho un solo paramentro

```
list.forEach(s -> System.out.println(s));
list.forEach(s -> copy.add(s));
list.forEach(s -> new Processor(s));
list.forEach(s -> Util.digest(s));
with
list.forEach(System.out::println); // method reference
list.forEach(copy::add); // method reference
list.forEach(Processor::new); // constructor reference
list.forEach(Util::digest); // (static) method reference
```

Esempio di codice dove uso una lambda espressione:

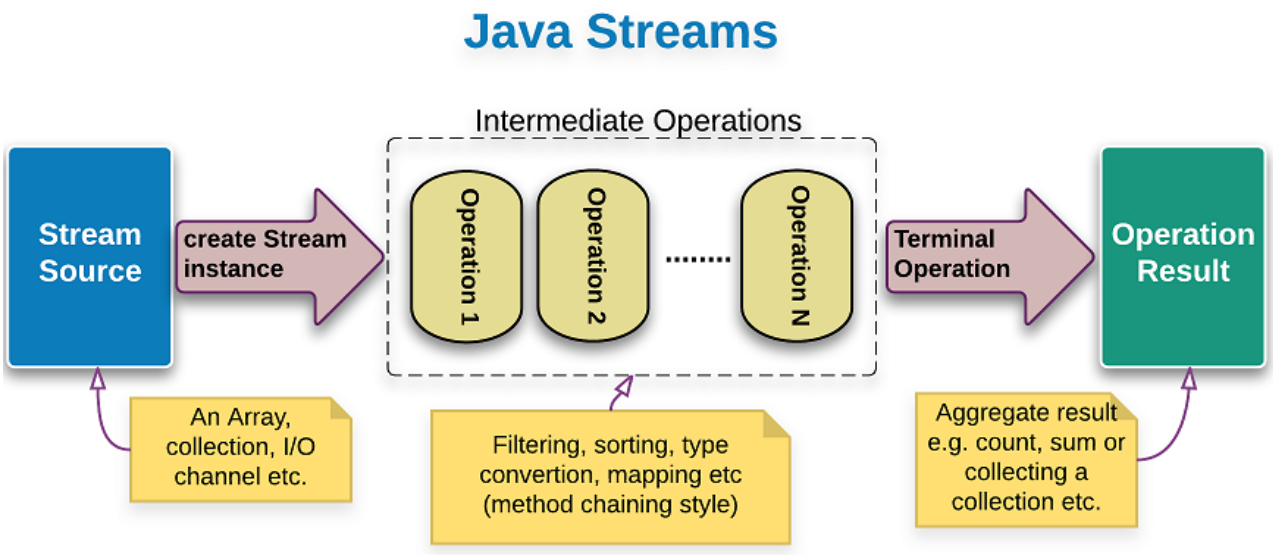
```
List<String> parole = Arrays.asList("casa", "albero", "gatto", "tavolo", "auto");

        // Utilizzo una lambda per filtrare le parole con length maggiore di 4 caratteri
        List<String> paroleLunghe = filtra(parole, s -> s.length() > 4);
```

**STREAM**

You can create streams from a collection c:

- → you can build the stream of its elements as `c.stream()` or `c.parallelStream()`
- → you have to import `java.util.stream.Stream<T>`




Quali possono essere le operazioni intermedie, vediamole:

- → `filter()` → Prende solo gli elementi che rispettano una condizione
  Ritorna un nuovo stream contenente gli elementi che soddisfano la condizione specificata

  Esempio d'uso, filtro i Santa Claus che provengono dalla Finlandia:

```
long howMany =
c.stream()
.filter(person -> person instanceof Santa)
.filter(person -> person.getCountry().equals("Finland"))
```

- → `map()` → Trasforma ogni elemento secondo una funzione specifica
  Ritorna un nuovo stream contenente i risultati dell'applicazione della funzione specificata a ciascun elemento

  Esempi d'uso:

```
.map(Person::getCountry)
.map(String::toUpperCase)

// Stream<Person>
.map(person -> (Santa) person) // Stream<Santa>
```

→ `flatmap()` → Appiattisce e trasforma gli elementi in un unico stream
Ritorna un nuovo stream dopo aver appiattito e trasformato gli elementi in un unico stream

Esempio d'uso:

```
.map(person -> (Santa) person) // Stream<Santa>
.map(Santa::getPresents) // Stream<Stream<String>>

Diventa:
.map(person -> (Santa) person) // Stream<Santa>
.flatMap(Santa::getPresents) // Stream<String>
```

→ `distinct()` → Rimuove duplicati dall'insieme
Ritorna un nuovo stream contenente solo gli elementi unici (senza duplicati)

Esempio d'uso, se i regali si ripetono elimino i duplicati:

```
.flatMap(Santa::getPresents) // Stream<String>
.distinct() // Stream<String>
```

→ `parallel()/ parallelStream()` → In Java, puoi ottenere uno stream parallelo utilizzando sia `parallelStream()` su una collezione, che `stream().parallel()` su uno stream già esistente. Entrambi i metodi producono lo stesso risultato: uno stream parallelo su cui è possibile eseguire operazioni in parallelo su più thread

Esempio d'uso:

```
Stream<Integer> parallelStream = numbers.stream().parallel();

// Operazione di trasformazione e stampa degli elementi parallela
parallelStream.map(n -> n * 2)
              .forEach(System.out::println);
```

→ `limit(n)` → Limita il numero di elementi
Ritorna un nuovo stream contenente al massimo i primi 'n' elementi

Esempio d'uso:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

// Creazione di uno stream dai numeri e limitazione a 5 elementi
Stream<Integer> limitedStream = numbers.stream().limit(5);

// Stampa degli elementi limitati
limitedStream.forEach(System.out::println);
```

→ `sorted()` → Ordina gli elementi
Ritorna un nuovo stream contenente gli elementi ordinati

Esempio d'uso, i regali di Santa Claus(stream<String>) li ordino in ordine alfabetico:

```
.flatMap(Santa::getPresents) // Stream<String>
.distinct() // Stream<String>
.sorted() // Stream<String>
```

→ `forEach()` → Esegue un'azione su ogni elemento
Non ritorna nulla esplicitamente, ma esegue un'azione su ogni elemento

Esempio d'uso, i regali ordinati ora li stampo:

```
.flatMap(Santa::getPresents) // Stream<String>
.distinct() // Stream<String>
.sorted() // Stream<String>
.forEach(System.out::println);
```

→ `forEachOrdered()` → è simile a `forEach()`, ma garantisce che l'operazione venga eseguita nell'ordine in cui gli elementi sono stati inseriti o nell'ordine naturale degli elementi, se applicabile
Non ritorna nulla esplicitamente, ma esegue un'azione su ogni elemento mantenendo l'ordine

Esempio d'uso:

```
// Stream di numeri
        Stream<Integer> numberStream = Stream.of(1, 2, 3, 4, 5);

// Utilizzo di forEachOrdered per sommare ogni numero con il suo
// doppio e stampare il risultato
        numberStream.forEachOrdered(n -> {
            int doubled = n * 2;
            int sum = n + doubled;
            System.out.println("Numero: " + n + ", Doppio: " +
doubled + ", Somma: " + sum);
        });
```

→ `anyMatch()` → Verifica se almeno un elemento soddisfa la condizione
Ritorna un valore booleano indicando se almeno un elemento soddisfa la condizione

Esempio d'uso, c'è almeno/esiste un Santa Claus che vive in Finlandia? True/False:

```
.filter(person -> person instanceof Santa) // Stream<Person>
.map(Person::getCountry) // Stream<String>
.anyMatch(country -> "Finland".equals(country))
```

→ `allMatch()` → Verifica se tutti gli elementi soddisfano la condizione
Ritorna un valore booleano indicando se tutti gli elementi soddisfano la condizione

Esempio d'uso:

```
boolean allAtLeast42 = c.stream().allMatch(s-> s.length() >=42);
```

→ `noneMatch()` → Verifica se nessun elemento soddisfa la condizione
Ritorna un valore booleano indicando se nessun elemento soddisfa la condizione

Esempio d'uso:

```
boolean noneAtLeast42 = c.stream().noneMatch(s->s.length()>=42);
```

→ `count()` → Conta il numero di elementi
Ritorna il numero totale di elementi nell'insieme

Esempio d'uso:

```
Collection<String> c = ... ;
//Count how many elements of c start with "a"
long howMany = c.stream()
.filter(s -> s.startsWith("a"))
.count();
```

→ `findAny()` → Ritorna un elemento arbitrario dell'insieme, se presente

Esempio d'uso:

```
Optional<String> found = c.stream()
.filter(s -> s.startsWith("a"))
.filter(s -> s.length() >= 42)
.findAny();
```

→ `findFirst()` → Ritorna il primo elemento dell'insieme, se presente

Esempio d'uso:

```
Stream<Persona> personeStream = Stream.of(
                new Persona("Mario", 25),
                new Persona("Luigi", 30),
                new Persona("Giovanna", 18),
                new Persona("Alessia", 20)
        );

        // Utilizzo di findFirst per trovare la prima persona
con un'età superiore a 18 anni
        Optional<Persona> primaPersonaMaggiorenne =
personeStream.filter(p -> p.getEta() > 18).findFirst();
```

**Implements ITERABLE**

Il forEach lo posso applicare solo agli iterabili, ovvero:

- array
- varargs
- collezioni
- oggetti di classi che implementano l'interfaccia `Iterable<T>`

E' importante ricordare che su array e collezioni posso applicare il forEach perché hanno il metodo `iterator()` dell'interfaccia `Iterable<T>` già implementato.
Questo significa che in certe situazioni io posso implementare `iterator()` dell'interfaccia `Iterable<T>` per ***delega*** richiamando `iterator()` di qualche costrutto che ce l'ha già implementato per default.

```
import java.util.Iterator;

public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // si può anche non implementare
}
```

Una classe su cui voglio poter iterare sulle sue istanze (applicare il forEach) deve essere iterabile.
Una classe che voglio far diventare iterabile deve implementare l'interfaccia `Iterable<T>`.
L'unico metodo che questa interfaccia contiene e che devo implemetare è `iterator()`.
Quello che deve ritornare il metodo `iterator()` dell'interfaccia `Iterable<T>` è un'implementazione concreta dell'interfaccia `Iterator<E>` e quindi un' implementazione concreta dei suoi metodi che sono:

- `boolean hasNext()`
- `E next()`.

Implementazione iterator() per delega:

```
public class Deck implements Iterable<Card> {
  private final SortedSet<Card> cards = new TreeSet<>();

  /**
   * Deck contiene un insieme di carte che sono SortedSet
   * SortedSet essendo una collezione ha già implementato per default
   * iterator(), quindi posso fare la delega
   */

  public Iterator<Card> iterator() {
      return cards.iterator();
  }
```

Esempi di implemetazioni di iterator():

```
public class Chat implements Iterable<Message> {
    private final Message[] messages = new Message[10];
    private int pos = 0;

public void add(Message message) {
    if (pos < 10)
       messages[pos++] = message;
}

    public Iterator<Message> iterator() {
       return new Iterator<Message>() {
          private int cursor = 0;

          public boolean hasNext() {
             return cursor < pos;
          }

          public Message next() {
             return messages[cursor++];
          }
       };
    }
}
```

```
public class Prefixes implements Iterable<String> {
    private final String s;

    public Prefixes(String s) {
       this.s = s;
    }

    public Iterator<String> iterator() {
       return new Iterator<String>() {
          private String ss = s;

          public boolean hasNext() {
             return ss != null;
          }

          public String next() {
             String result = ss;
             if (ss.length() > 0)
                ss = ss.substring(0, ss.length() - 1);
             else
                ss = null;
             return result;
          }
       };
    }
}
```

```
public class Primes implements Iterable<Integer> {

    public Iterator<Integer> iterator() {
       return new Iterator<Integer>() {
          private int next = 2;

          public boolean hasNext() {
             return true;
          }

          public Integer next() {
             int result = next;
             advance();
             return result;
          }

          private void advance() {
             for (next++; !isPrime(next); next++);
          }

          private boolean isPrime(int x) {
             for (int d = 2; d < x; d++)
                if (x % d == 0)
                   return false;

             return true;
          }
       };
    }
}
```

```
@Override public final Iterator<VotiPerPartito> iterator() {
       return new Iterator<VotiPerPartito>() {
          private final Iterator<Partito> it = new TreeSet<>(voti.keySet()).iterator();

          @Override public boolean hasNext() {
             return it.hasNext();
          }

          @Override public VotiPerPartito next() {
             Partito prossimoPartito = it.next();
             return new VotiPerPartito(prossimoPartito, votiPer(prossimoPartito));
          }

       };
}
```

```
/**
 *Si renda PunishableSet iterabile, in modo che iterando su un tale
 *oggetto si ottengano gli elementi dentro l'insieme (si tratta di
 * aggiungere un solo metodo public).
 */

@Override
public Iterator<E> iterator() {
return container.keySet().iterator();
```

## ALTRO

Random: generare valori casuali vincolati però ad appartenere ad un preciso intervallo:

```
// genero un array casuale di lettere alfabetiche minuscole
char[] arr = new char[length];
for (int pos = 0; pos < length; pos++)
    arr[pos] = (char) ('a' + random.nextInt('z' - 'a'));
```

```
// generiamo una stringa casuale lunga da 1 a 5 caratteri
int len = 1 + random.nextInt(5);
String s = "";
while (len-- > 0)
     s += (char) ('a' + random.nextInt(26));
```

Utilizzo combinato di concatenazione tra stringe, charAt e modulo:

```
protected String barra(int i, double frequenza) {
     String barra = "";
     for (int j = 1; j <= frequenza; j++)
         barra += "*@+".charAt(i % 3);
     return barra;
}
```

Creazione di una lista con tutte le parole di un file(utilizzo di split()): uso come delimitatore i caratteri non alfabetici

```
try(BufferedReader r = new BufferedReader(new FileReader(filename))){
     while((line = r.readline()) != null)
          for(String s : line.split("\\W+"))
               if(selector.test(s))
                    add(s);
}
```

## Metodo collect()

Sintassi:

```
Collezione collect(Collectors.metodoStaticoDiCollectors)
```

Da importare la classe `Collectors` nel codice per utilizzare i suoi metodi statici.

```
import java.util.stream.Collectors;
```

Il metodo `collect` viene utilizzato per trasformare gli elementi di uno stream in una struttura dati diversa, ad esempio una lista, un set o una mappa. Accetta un oggetto `Collector` che definisce come aggregare gli elementi.

Esempio:

```
List<String> strings = Arrays.asList("a", "b", "c", "d");
List<String> collectedList = strings.stream()
                                    .map(String::toUpperCase)
                                    .collect(Collectors.toList());
System.out.println(collectedList); // Output: [A, B, C, D]
```

Altri metodi sttatici tipici di Collectors che puoi usare :

```
.collect(Collectors.toSet());
.collect(Collectors.toMap(String::toUpperCase, String::length))
.collect(Collectors.joining(", "))
.collect(Collectors.groupingBy(String::length))
.collect(Collectors.partitioningBy(s -> s.length() > 2))
```

---

## Metodo entrySet()

La funzione `entrySet()` è un metodo molto utile che fa parte dell'interfaccia `Map` in Java.

## Cosa fa `entrySet()`:

Il metodo `entrySet()` restituisce un insieme (Set) di oggetti `Map.Entry`. Ogni oggetto `Map.Entry` rappresenta una coppia chiave-valore all'interno della mappa. In pratica, `entrySet()` consente di ottenere una vista della mappa sotto forma di un insieme di coppie chiave-valore, dove ciascuna coppia è rappresentata da un oggetto `Map.Entry`.

## Come può essere utilizzato:

1. **Iterazione sugli elementi della mappa**:

Puoi utilizzare `entrySet()` per iterare su tutti gli elementi di una mappa e ottenere sia la chiave che il valore di ciascuna coppia chiave-valore:

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
}
```

2. **Modifica degli elementi della mappa durante l'iterazione**:

Puoi utilizzare gli oggetti `Map.Entry` ottenuti da `entrySet()` per modificare gli elementi della mappa durante l'iterazione:

```
java
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
    if (entry.getValue() == 2) {
        entry.setValue(20);
    }
}

System.out.println(map); // Output: {one=1, two=20, three=3}
```

3. **Conversione della mappa in altri tipi di collezioni**:

Puoi utilizzare `entrySet()` per convertire la mappa in altri tipi di collezioni, ad esempio una lista di coppie chiave-valore:

```
java
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);

List<Map.Entry<String, Integer>> entryList = new ArrayList<>(map.entrySet());
```

In sintesi, `entrySet()` è un metodo molto utile per accedere e manipolare le coppie chiave-valore all'interno di una mappa in Java. È particolarmente utile quando si desidera iterare sugli elementi di una mappa e ottenere sia la chiave che il valore di ciascuna coppia.

Puoi utilizzare `entrySet()` in uno stream per eseguire operazioni specifiche sugli elementi della mappa, ad esempio filtrarli, trasformarli o raccoglierli in una nuova struttura dati. Ecco alcuni esempi di come potresti utilizzarlo in uno stream:

1. **Filtrare le coppie chiave-valore in base a un criterio**:

Supponiamo di avere una mappa di persone con il loro nome e la loro età, e vogliamo ottenere solo le persone con età superiore a 18 anni:

```
Map<String, Integer> people = new HashMap<>();
people.put("Alice", 25);
people.put("Bob", 30);
people.put("Charlie", 17);

List<Map.Entry<String, Integer>> adults = people.entrySet().stream()
    .filter(entry -> entry.getValue() > 18)
    .collect(Collectors.toList());

System.out.println(adults);
```

2. **Trasformare le coppie chiave-valore in un'altra forma**:

Supponiamo di voler trasformare le coppie chiave-valore della mappa in una lista di stringhe nel formato "Nome: Età":

```
Map<String, Integer> people = new HashMap<>();
people.put("Alice", 25);
people.put("Bob", 30);
people.put("Charlie", 17);

List<String> formattedList = people.entrySet().stream()
    .map(entry -> entry.getKey() + ": " + entry.getValue())
    .collect(Collectors.toList());

System.out.println(formattedList);
```

3. **Calcolare un valore aggregato basato sulle coppie chiave-valore**:

Supponiamo di voler calcolare la somma di tutte le età delle persone nella mappa:

```
Map<String, Integer> people = new HashMap<>();
people.put("Alice", 25);
people.put("Bob", 30);
people.put("Charlie", 17);

int totalAge = people.entrySet().stream()
    .mapToInt(Map.Entry::getValue)
    .sum();

System.out.println("Total age: " + totalAge);
```

In questi esempi, `entrySet()` viene chiamato sulla mappa per ottenere una vista delle coppie chiave-valore come un insieme, che può quindi essere utilizzato all'interno di uno stream per eseguire varie operazioni.