Esempio

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

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

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

// Output: Luigi Sofia
```

- `filter()` operazione intermedia che restituisce gli elementi che soddisfano il predicato passato
- `sorted()` operazione intermedia **stateful** che restituisce uno stream che ha gli elementi ordinati in base al Comparator passato
- `comparing()` permette di estrarre la chiave per il confronto
- `forEach()` operazione terminale che esegue un’azione su ciascun elemento dello stream (su uno stream parallelo l’ordine non è garantito)

Esempio

- Data una lista di istanze di Persona trovare i diversi ruoli

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

// Output: CTO Programmer GrLeader
```

- `distinct()` operazione intermedia **stateful** che restituisce uno stream di elementi distinti
- Data una lista di istanze di Persona trovare il nome di una che ha il ruolo Programmer

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

// Output: Luigi
```

- `findAny()` (simile a `findFirst()`) operazione terminale che restituisce un `Optional`, per valutarla non è necessario esaminare tutto lo stream, si dice short-circuiting (può far sì che alcune parti non eseguano)

Stateless–Stateful

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

Generare Stream: `iterate()`

- Gli stream possono essere generati a partire da una funzione tramite le operazioni `iterate()` e `generate()`. Queste operazioni creano stream infiniti
- `iterate()` restituisce un stream infinito e ordinato dato dall’esecuzione iterativa di un funzione `f` applicata inizialmente ad un elemento `seme`, quindi produce uno stream di `seme`, `f(seme)`, `f(f(seme))`, etc.
- `limit()` tronca lo stream ad una lunghezza pari al numero indicato

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

- Il codice sopra dà lo stream di 10 elementi che consiste in 2, 4, 8, … 1024
- Ad ogni passo, dopo il primo, il valore in input alla funzione è il valore calcolato dalla funzione al passo precedente
- L’operazione `iterate()` è sequenziale

Generare Stream: generate()

- Il metodo generate() permette di produrre uno stream infinito di valori, tramite una funzione di tipo Supplier, ovvero che fornisce un valore
  - generate() non applica una funzione ad ogni nuovo valore prodotto, come invece fa iterate()

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

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

Tipo IntStream

- IntStream rappresenta uno stream di valori int

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

- Il codice sopra genera e stampa i quadrati dei numeri da 1 a 6
- rangeClosed() restituisce una sequenza di int nell'intervallo specificato (estremi inclusi) con incremento pari a 1
- sum() somma i valori presenti nello stream IntStream

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


Tipo IntStream

```
int result =
 Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
       .mapToInt(x -> x.length()).sum();
// Output: result = 31
```

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

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

- boxed() restituisce uno Stream di Integer a partire da un IntStream
- Si abbia la lista che contiene istanze di Persona, si generi uno stream contenente i primi 4 elementi

```
List<Persona> lista;

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

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

Debug

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

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
       .map(x -> x + 17)
       .filter(x -> x % 2 == 0)
       .limit(3)
       .forEach(System.out::println);

// Output: 20 22
```

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

Debug Con Peek

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

// Output:
// from stream: 2
// after map: 19
// from stream: 3
// after map: 20
// after filter: 20
// after limit: 20
// from stream: 4
// after map: 21
// from stream: 5
// after map: 22
// after filter: 22
// after limit: 22
```


Esercizio 1 Con Java 8

- Data una lista di stringhe
  - Esempio: {“author”, “auto”, “autocorrect”, “begin”, “big”, “bigger”, “biggish”}
  - Produrre una lista che contiene solo le stringhe che cominciano con un certo prefisso noto
  - Esempio: se il prefisso è “au”, la lista prodotta è {“author”, “auto”, “autocorrect”}
- Suggerimento
  - Usare il metodo `substring(int beginIndex, int endIndex)` della classe `String` che restituisce la sottostringa che inizia a `beginIndex` e termina a `endIndex`
  - Esempio: “ciao”.`substring(0, 2)` restituisce “ci”

Esercizio 2 Con Java 8

- Data una lista di stringhe
  - Esempio: {“to”, “speak”, “the”, “truth”, “and”, “pay”, “your”, “debts”}
  - Produrre una stringa contenente le iniziali di ciascuna stringa della lista
  - Esempio: per la lista sopra si produrrà la stringa “tsttapyd”

Esercizio 3 Con Java 8

- Data una lista di terne di numeri interi
  - Per ciascuna terna verificare se essa costituisce un triangolo
  - Restituire la lista dei perimetri per le terne che rappresentano triangoli
- Suggerimenti
  - In un triangolo, ciascun lato è minore della somma degli altri due
  - Si può rappresentare la terna come un array di tre elementi interi

```
int[] t = new int[] { 2, 3, 4 };
```

  - Si può rappresentare la lista di terne come lista di array di interi

```
List<int[]> lista;
```

Esercizio 3

- Soluzione

```
private List<int[]> terne = Arrays.asList(new int[] { 2, 2, 3 },
   new int[] { 3, 2, 3 }, new int[] { 3, 3, 3 }, new int[] { 3, 4, 5 },
   new int[] { 5, 2, 3 });


private List<Integer> verifica() {
   return terne.stream()
         .filter(t -> t[0]<t[1]+t[2])
         .filter(t -> t[1]<t[0]+t[2])
         .filter(t -> t[2]<t[0]+t[1])
         .map(t -> t[0] + t[1] + t[2])
         .collect(Collectors.toList());
}
```


Esercizio 4

- Data una lista di numeri interi
  - Verificare se ciascuna terna formata prendendo dalla lista tre numeri contigui costituisce un triangolo
  - Esempio: lista {2, 3, 5, 7, 8}, terne {2, 3, 5}, {3, 5, 7}, {5, 7, 8}
  - Restituire la lista delle terne che rappresentano triangoli
  - Esempio: terne {3, 5, 7}, {5, 7, 8}
- Suggerimenti
  - In un triangolo, ciascun lato è minore della somma degli altri due
  - Si generano gli indici da 0 a n-2
  - Si conserva in un array di tre elementi, per ciascun indice i, l'elemento i, ed i due successivi elementi, così da formare una terna

Esercizio 4

- Soluzione

```
private List<Integer> lista = List.of(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);

private List<int[]> verifica() {
 return IntStream.rangeClosed(0, lista.size() - 3)
   .mapToObj(i -> new int[]{lista.get(i), lista.get(i+1), lista.get(i+2)})
   .filter(t -> t[0] < t[1] + t[2])
   .filter(t -> t[1] < t[0] + t[2])
   .filter(t -> t[2] < t[0] + t[1])
   .collect(Collectors.toList());
}
```


Esercizio 5

- Data una lista di numeri interi positivi
  - Verificare se la lista è ordinata
- Suggerimenti
  - Si generano gli indici da 0 a n-1
  - Per ciascun valore dell'indice i, si confrontano l'elemento con indice i ed il successivo, se il secondo è minore del primo la lista non è ordinata e si può fermare la verifica

Esercizio 5

- Soluzione 1
  - Ogni iterazione accede a due elementi della lista
  - L'operazione `filter` emette l'indice `i` dell'elemento che è più grande del successivo
  - Appena `filter` trova un elemento, con l'operazione `findAny` ferma la ricerca
  - **Nessuna operazione conserva uno stato globale**

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 12, 3);
private boolean isOrdinata() {
  return IntStream.rangeClosed(0, lista.size() - 2)
    .filter(i -> lista.get(i) > lista.get(i+1))
    .peek(v -> System.out.print(lista.get(v) + " > " + lista.get(v+1)))
    .findAny()
    .isEmpty();
}
// 12 > 3
```


Esercizio 5

- Soluzione 2
  - Si conserva uno stato che è condiviso fra varie iterazioni

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);
private int prec; // conserva l'elemento precedente, e' lo stato condiviso
private boolean isOrdinata() {
   prec = lista.get(0);
   return lista.stream()
      .filter(v -> seMinoreDiPrec(v)) // modifica prec, scarta valori false
      .findAny() // si ferma quando vi e' un false
      .isEmpty();
}
private boolean seMinoreDiPrec(int x) { // non puo' eseguire in parallelo
   int p = prec;
   prec = x; // modifica lo stato
   return x < p; // ritorna true se elemento corrente > elemento prec
}
```


Considerazioni

- Per la soluzione 1, si ha

```
.filter(i -> lista.get(i) > lista.get(i+1))
```

- l'espressione lambda passata a `filter` legge due numeri dalla lista e dà in output un `boolean`, senza altri effetti collaterali (side-effect-free), ovvero **non modifica uno stato condiviso**. Tale espressione lambda è una **funzione pura**
- Per la soluzione 2, si ha

```
.filter(v -> seMinoreDiPrec(v))
```

ovvero

```
.filter(v -> { int p = prec; prec = v; return v < p; })
```

- l'espressione lambda passata a `map` modifica un attributo. Tale modifica è un effetto collaterale (voluto), quindi non è una funzione pura
- Si noti che: (i) `prec` è un attributo, definito al di fuori dell'espressione lambda che può essere acceduto da essa (accessi al contesto sono consentiti); (ii) la modifica di uno stato condiviso non permette l'esecuzione parallela