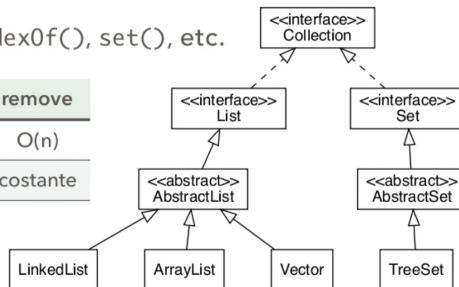


- Le librerie di Java hanno tante interfacce e classi **collection**. Collection è un'interfaccia che definisce i metodi add(), remove(), size(), contains(), containsAll(), etc.
- List definisce i metodi get(), indexOf(), set(), etc.

	get	add	contains	remove
ArrayList	costante	costante	O(n)	O(n)
LinkedList	O(n)	costante	O(n)	costante
TreeSet			O(lg n)	O(lg n)



Il metodo stream() permette di accedere alla programmazione funzionale a partire da un sottotipo di Collection, restituendo un tipo Stream che ha metodi che prendono in ingresso funzioni lambda. Collection mette a disposizione il metodo stream() che rappresenta un'eccezione per l'interfaccia, esso è un metodo di default. Un metodo di default non può agire sullo stato in quanto l'interfaccia non ne ha uno.

- filter()**: operazione lazy o **non terminale** che seleziona alcuni elementi sulla base della condizione della funzione lambda. Filter prende in input un tipo Predicate e se l'espressione non torna un booleano allora c'è un errore di compilazione. La funzione viene applicata a tutti gli elementi della lista. Filter restituisce un tipo Stream che contiene solo gli elementi che soddisfano il Predicate.

```

List<String> nomi = List.of("Nobita", "Nobi", "Suneo"); // crea la lista
long c = nomi.stream().filter(s->s.equals("Nobi")).count();
  
```

- count()**: conta gli elementi presenti nello Stream. Questa è un'operazione **terminale** che fa finire forzatamente le operazioni lazy (intermedie) precedenti come la filter. Uno Stream non viene riempito fino a quando non si hanno operazioni terminali. Le operazioni lazy tornano tipi Stream.

```

long c = nomi.stream()
    .filter(s -> s.length() == 5)
    .count();
  
```

- reduce()**: è un'operazione **terminale** che applicata ad uno Stream fornisce un unico risultato calcolato sull'intero Stream. Lo vauta internamente sulla base dell'espressione lambda passatagli come parametro. Reduce() prende in input un T identity ed un BinaryOperator<T> accumulator e calcola l'operazione binaria degli elementi della lista a partire dal parametro identity con il primo elemento della lista, prosegue applicandola tra il risultato dell'operazione e al secondo elemento e così via.

```

reduce(T identity, BinaryOperator<T> accumulator)

Predicate<Integer> positive = x -> x>=0;
Stream<Integer> result = Stream.of(2,5,10,-1).filter(positive);
reduce(0, (accum,v) -> accum+v); // ritorna 2+5+10=17
  
```

Alla prima iterazione accum=0 perchè è l'elemento specificato, v corrisponde al generico valore nello stream e l'espressione lambda prende i due parametri e mette ad ogni iterazione il risultato in accum. Con reduce() passiamo da un'insieme di valori ad un singolo valore. In reduce è possibile passare un metodo di una determinata classe :

```
reduce(0, Integer::sum); // -> restituisce T  
reduce(Integer::sum) // -> restituisce Optional<T>. Invocare isPresent()
```

All'interno di Integer c'è il metodo sum() e rappresenta un'espressione lambda dato che reduce prende in input solo funzioni lambda. La versione che non prende il valore iniziale di accum restituisce un tipo Optional<T> del quale bisogna verificare l'esistenza tramite il metodo isPresent(). infatti Optional è un contenitore che potrebbe non avere un risultato.

- Se lo Stream è vuoto anche Optional sarà vuoto poichè non ha elementi su cui effettuare l'operazione.
- Se lo Stream contiene un solo elemento allora verrà ritornato l'unico elemento.

Sul tipo Optional restituito dalla reduce si invocano i metodi isPresent() ed isEmpty() per verificare se è pieno o vuoto l'oggetto restituito.

```
// Esempio uso reduce()  
public class Pagamenti {  
    private List<Float> importi = new ArrayList<>();  
  
    // in stile imperativo  
    public float calcolaSommaImper() {  
        float risultato = 0f;  
        for (float v : importi)  
            risultato += v;  
        return risultato;  
    }  
  
    // in stile funzionale  
    public float calcolaSomma() {  
        return importi.stream().reduce(0f, Float::sum);  
    }  
}
```

- **map()** : si applica ad uno Stream e prende in input una espressione Lambda. Produce uno Stream in uscita (per tanto è un'operazione **non terminale**) i cui valori sono differenti da quelli dello Stream in input.

```
map(Function<T,R> mapper); // T è il tipo in input. R è il tipo in output  
  
// Esempio  
List<Integer> l = List.of(1,2,5);  
Stream<Integer> s1 = l.stream().map(x -> x * 2); // risultato = [2,4,10]  
List<Integer> s2 = s1.toList(); // converte in List uno Stream  
  
-----  
List<Persona> l = List.of(new Persona("Pippo", 46), new Persona("Alessio", 18));  
Stream<Integer> result = l.stream().map(Persona::getEta); // result = [46,18]  
// l.stream().map(p -> p.getEta()); // chiamata equivalente  
  
map(Persona::getEta); // Per ogni elemento dello Stream si chiama il metodo getEta present  
in Persona.
```

Calcolare la somma delle età delle persone di una lista di oggetti Persona

```
List<Persona> amici = Arrays.asList(  
    new Persona("Saro", 24), new Persona("Taro", 21),  
    new Persona("Ian", 19), new Persona("Al", 16));  
  
// somma calcolata in stile funzionale con i riferimenti ai metodi  
int somma = amici.stream()  
    .map(Persona::getEta)  
    .reduce(0, Integer::sum);  
  
• La funzione passata a map() è il metodo getEta() di Persona
```

Ricerca del massimo in stile funzionale :

1) Dato il seguente metodo nella classe persona

```
public static Persona getMax(Persona p1, Persona p2) {  
    if (p1.getEta() > p2.getEta())  
        return p1;  
    return p2;  
}
```

```
Optional<Persona> pmax = amici.stream()  
    .filter(x -> x.getEta() < 20)  
    .reduce(Persona::getMax);  
  
if (pmax.isPresent())  
    System.out.println("persona: " + pmax.get().getNome());
```

2) Sfrutto Comparator.comparing() :

```
Optional<Persona> pmax = amici.stream()  
    .filter(x -> x.getEta() < 20)  
    .max(Comparator.comparing(Persona::getEta));  
  
if (pmax.isPresent())  
    System.out.println("persona: " + pmax.get().getNome());
```

• **max()** è un operazione **terminale** che restituisce, a partire da uno Stream, il valore massimo .
L'operazione max() prende un **Comparator** (un **interfaccia funzionale**) e restituisce un Optional, operando in modo simile a reduce() . Comparator.**comparing()** prende in input una funzione di confronto che estrae una chiave e restituisce un Comparator. Comparator implementa una funzione **compare** che stabilisce l'ordinamento di una collezione di oggetti, e max() chiama il metodo compare() del comparator in input.

Comparator.comparing(Persona :: getEta) —> il parametro di confronto in comparing() è l'età
comparing() —> al suo interno chiama il metodo compare() e fa un confronto fra due parametri

- **collect()** : permette di raggruppare i risultati e prende in ingresso un tipo **Collector**. La classe Collectors implementa metodi utili per raggruppamenti, come ad esempio il metodo `toList()` che accumula elementi in una List.

```
List<Persona> amici = List.of(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```

- Ricaviamo la lista delle età

```
List<Integer> e = amici.stream()
    .map(x -> x.getEta())
    .collect(Collectors.toList());
```

`collect(Collectors.toList())` trasforma in List ma in generale `collect()` permette di trasformare in qualsiasi cosa.

Programmazione parallela

Un flusso parallelo ci consente di utilizzare l'elaborazione multi-core eseguendo l'operazione di flusso **in parallelo** su più core della CPU. I dati vengono suddivisi in più flussi secondari e l'operazione prevista viene eseguita in parallelo e, infine, i risultati vengono aggregati per formare l'output finale.

Un flusso creato in Java è sempre di natura seriale per impostazione predefinita, se non diversamente specificato. Possiamo convertire il flusso in un flusso parallelo in due modi:

- possiamo invocare `Collections.parallelStream()`
- possiamo invocare `Stream.parallel()`

```
List<Integer> list = Arrays.asList( 1, 2, 3, 4, 5 );~
//Esecuzione seriale~
list.stream().forEach( a -> System.out.println(a + ":" + Thread.currentThread().getName()) );~
//parallel()~
list.stream().parallel().forEach( a -> System.out.println(a + ":" + Thread.currentThread().getName()) );~
//parallelStream~
list.parallelStream().forEach(number -> System.out.println(number + ":" + Thread.currentThread().getName()) );~
```

- In generale, si produce sempre uno Stream (output) differente dallo Stream in input
- Tutto quello che avviene nello Stream iniziale può avvenire separatamente rispetto a quello che si produce in output e ciò **garantisce la CORRETTEZZA** visto che l'input non viene modificato in alcun modo (in particolar modo quando uso il parallelismo)
- I programmatore potrebbero sbagliare nello scrivere il codice:
- L'esecuzione in parallelo non prevede di avere uno stato globale e quindi non si aggiorna uno **stato globale** e in questo caso il parallelismo va benissimo
- In caso contrario, quindi **si modifica uno stato globale**, ci si deve chiedere se è necessario farlo. Se serve farlo, bisogna valutare e controllare di farlo correttamente, controllando anche l'ordine delle operazioni (diventa complicato).
- Per stato globale si intende una qualsiasi **variabile condivisa** tra diversi flussi d'esecuzione.
- Le prestazioni migliorano in caso di parallelismo (basta inserire `.parallel()`)
- Se uso un `LinkedList` ci perdo rispetto all'uso di un `ArrayList` nei tempi di accesso ad un elemento poiché gli elementi in `ArrayList` sono indicizzati e permettono tempo di accesso costante.
- Se si usano i `parallelStream()` si possono avere dei guadagni in termini di tempo e prestazioni man mano che l'hardware si aggiorna
- Il metodo `parallelStream()` **POSSIBILMENTE** da uno Stream parallelo perché si deve valutare se il parallelismo è supportato dall'hardware prima di avviare tale procedura. Viene valutata anche la dimensione lo Stream e la sua dimensione: se è ridotta allora viene scandita in maniera sequenziale perché ci vuole meno tempo rispetto all'avvio del parallelismo stesso

- **sorted()** : è un'operazione **non terminale** STATEFUL (deve conoscere tutti gli elementi dello Stream) che a partire da uno Stream restituisce uno Stream nel quale gli elementi sono ordinati rispetto ad una relazione d'ordine passata tramite un Comparator. Altre operazioni come **min()**, **max()** si aspettano un Comparator nello stesso formato per poter lavorare.
- **forEach()** : è un'operazione **terminale** e permette di eseguire un'operazione per ogni elemento dello Stream(). Essa non ha un valore di ritorno ed in più distrugge lo Stream di partenza sulla quale viene applicata. Si nota che dopo forEach() non possono quindi essere invocate delle operazioni sullo Stream in quanto non esiste più.

- 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() + " "));
```

- **distinct()** : è un'operazione **non terminale** stateful che restituisce uno Stream() senza duplicati.

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

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

- **findAny()** : è un'operazione **terminale** che restituisce un Optional. Essa permette di fare short-circuiting in uno Stream, ovvero di analizzarlo non per intero, infatti restituisce qualunque elemento a partire da un flusso filtrato.
- **findFirst()** : identico a findAny() ma restituisce il primo elemento dello Stream filtrato.

- 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());
```

- 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**

- **iterate()** : è un’operazione che produce uno Stream infinito seguendo una funzione lambda presa in input. Si invoca l’operazione **limit()** per limitare il numero di volte che deve essere chiamata iterate(). L’operazione iterate() prende in input un elemento iniziale ed un’espressione lambda (compatibile col tipo di elemento passato).

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

- **generate()** : è un’operazione che permette di produrre uno Stream infinito di valori a partire da una funzione che fornisce un valore. Anche generate() va bloccata con l’utilizzo di limit.

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

limit() può essere usata anche con filter() per limitare un numero massimo di elementi che “passano” il controllo.

IntStream : rappresenta uno Stream di soli valori interi che mette a disposizione vari metodi non compatibili con Stream<Integer> .

- **rangeClosed(a, b)** : produce un IntStream che comprende valori interi compresi fra gli estremi indicati (inclusi).
- **sum()** : somma gli elementi in uno IntStream e ritorna un valore intero.
- **mapToInt()** : esegue la funzione lambda presa in input **su uno Stream** e restituisce un IntStream

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

- **boxed()** : restituisce uno Stream<Integer> a partire da un IntStream.

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

- **mapToObj()** : è un metodo che costruisce uno Stream di Oggetti a partire da un IntStream .

```
public static void main(String[] args) {
    IntStream intStream = IntStream.range(7, 15);
    Stream<String> s = intStream.mapToObj(a -> Integer.toBinaryString(a));
    s.forEach(System.out::println);
}
```

- **peek()** : non modifica lo Stream su cui opera ed è un'operazione che solitamente si usa solo per il debug. Quindi è possibile mostrare come viene modificato lo Stream durante le operazioni.

```
List 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());
```

Function <T, R>

Una map() prende in ingresso una Function<T, R>, ovvero prende in input un tipo T e torna un tipo R. Function è un'interfaccia funzionale che definisce l'unico metodo **apply()**.

```
Function<String, Integer> stringLength = x -> x.length();

int result = Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature") // ->
Stream<String>
    .map(stringLength) // -> Stream<Integer>
    .reduce(0, Integer::sum); // 31
```

Supplier <T>

Supplier è un'interfaccia funzionale che ha un singolo metodo chiamato **get()**. Non pende in input nessun parametro e restituisce un unico valore di tipo T.

```
Supplier sup = () -> "ciao ciao";
String s = sup.get(); // s: ciao ciao
```

Si nota che sup non contiene un oggetto String ma il codice che lo genera, ovvero un'espressione Lambda.
Si può implementare in maniera concisa il Factory Method con l'utilizzo di un Supplier.

Senza supplier :

```
Prodotto p = Creator.getProdotto("primo");

public class Creator {
    public static Prodotto getProdotto(String name) {
        switch (name) {
            case "primo": return new ProdottoA();
            case "secondo": return new ProdottoB();
            case "terzo": return new ProdottoC();
            case "quarto": return new ProdottoD();
            default: return new ProdottoA();
        }
    }
}
```

Con l'uso di Supplier si evitano costrutti condizionali :

- Usando un Supplier

```
Supplier<Prodotto> prodSupplier = ProdottoA::new;
```
- La linea di codice sopra è equivalente a

```
Supplier<Prodotto> suppl = () -> new ProdottoA();
```
- Per avere istanze di sottotipi di Prodotto, si chiama get() sul Supplier

```
Prodotto p1 = prodSupplier.get();
```
- Quindi si crea una mappa che fa corrispondere al nome di un Prodotto la sua creazione

```
Map<String, Supplier<Prodotto>> map = Map.of("primo", ProdottoA::new, "secondo", ProdottoB::new, "terzo", ProdottoC::new);
```
- Si usa la mappa per istanziare sottotipi di Prodotto

```
public static Prodotto getProdotto(String name) {  
    Supplier<Prodotto> s = map.get(name);  
    if (s != null)  
        return s.get();  
    return new ProdottoA();  
}
```
- Il frammento di codice sopra è equivalente a

```
public static Prodotto getProdotto(String name) {  
    return map.getOrDefault(name, ProdottoA::new).get();  
}
```

Si usa un Supplier per ogni ConcreteProduct che servono.

Si può usare un metodo della Map stessa nel seguente modo:

```
public static Prodotto getProdotto(String name) {  
    return map.getOrDefault(name, ProdottoA::new).get();  
}
```

- Il metodo **getOrDefault()** ritorna il valore riferito alla chiave passata `name` altrimenti, se non esiste, ritorna un **default** definito dal programmatore