Java8 Features

By

Praveen Oruganti

Blog: https://praveenoruganti.blogspot.com

Facebook Group: https://www.facebook.com/groups/2340886582907696/

Github repo: https://github.com/praveenoruganti

Java 8 Features

1. Lambda Expression
2. Functional Interface
3. Method Reference
4. Stream API
5. New Date-Time API

1.Lambda Expression

It facilitates functional programming and it came up as an alternative for annonymous function.
It is applicable only for functional Interface and it reduces the boiler plate code.
(parameters) -> expression;
Or
(parameters) -> {statements};

1. **With no parameter**

```
Runnable r= () -> System.out.println("This in run method") ;
Thread t1 = new Thread(r);
t1.start();
```

2. **With single parameter**

```
interface DisplayInterface{
void display(String name);
}

DisplayInterface di= (name) -> System.out.println(name) ;
di.display("Praveen") ;
```

3. **With multiple parameters**

```
interface Calculator {
int add(int num1, int num2);
}
Calculator ci1= (int num1,int num2) -> {
    int sum=num1+num2;
    System.out.println(sum);
    return sum;
};
System.out.println(ci1.add(5, 6));
```

2.Functional Interface

It is an Interface which has only one abstract method and can have any number of default and static methods.

It needs to be represented with the help of annotation @FunctionalInterface.

For example

```java
@FunctionalInterface
interface Interface1 {
    void show();

    default void display() {
        System.out.println("Interface1 Display");
    }

    static void display1() {
        System.out.println("Interface1 Display1");
    }
}
```

<u>What is the need of default method in an interface?</u>

We can provide default functionality with default method.
So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

For example, i have 2 similar default methods in 2 interfaces and a class implementing those 2 interfaces will get an compile error like duplicate default methods so in order to avoid that we need to override the same method in the class as well.

For example,

```java
@FunctionalInterface
interface Interface1 {
    void show();

    default void display() {
        System.out.println("Interface1 Display");
    }
    static void display1() {
        System.out.println("Interface1 Display1");
    }
}

@FunctionalInterface
interface Interface2 {
    void print();

    default void display() {
        System.out.println("Interface2 Display");
    }

    static void display1() {
        System.out.println("Interface2 Display1");
    }
}
```

```java
public class FunctionalInterfaceTest implements Interface1, Interface2 {

    @Override
    public void print() {
        System.out.println("Print");
    }

    @Override
    public void show() {
        System.out.println("Show");
    }

    @Override
    public void display() {
        System.out.println("Class Display");
    }

    public static void main(String[] args) {
        Interface1.display1();
        Interface2.display1();
    }

}
```

What is the need of static method in an interface?

Java 8 introduced static methods to provide utility methods on interface level without creating the object of the interface.
For example,

- ✓ Stream.of()
- ✓ Stream.iterate()
- ✓ Stream.empty()

Please refer the above example for static methods in interface which we discussed.

Predefined Functional Interfaces

a. Predicate -> test()
b. Function -> apply()
c. Consumer -> accept()
d. Supplier -> get()

a.Predicate

It accepts an object as input and returns a boolean i.e.. true or false.
Predicate chaining is also possible and we can use and(), or(),negate()

For example,

```java
/* Predicate predefined functional interface */
Predicate<Integer> pi1= i -> (i > 10);
Predicate<Integer> pi2= i -> (i < 100);
System.out.println("pi1.test(100) "+ pi1.test(100));
System.out.println("pi1.test(7) " + pi1.test(7));
System.out.println("pi2.test(100) " +pi2.test(100));
System.out.println("pi2.test(99) " +pi2.test(99));

Predicate<String> ps = s -> (s.length() > 3);
System.out.println("ps.test(\"Praveen\") " + ps.test("Praveen"));
System.out.println("ps.test(\"op\") "+ ps.test("op"));

Predicate<Collection> pc = c ->  c.isEmpty();
List al= new ArrayList();
System.out.println("pc.test(al) " +pc.test(al));
al.add("Praveen");
System.out.println("pc.test(al) " +pc.test(al));

/*predicate joining is also possible with and(), or(), negate() */

Predicate<Integer> pi1pi2 = pi1.and(pi2);
System.out.println("pi1pi2.test(101) " + pi1pi2.test(101));
System.out.println("pi1pi2.test(9) " + pi1pi2.test(9));
System.out.println("pi1pi2.test(11) " + pi1pi2.test(11));
System.out.println("pi1pi2.test(99) " + pi1pi2.test(99));

Predicate<Integer> pi1pi2D= i -> (i > 10) && (i < 100) ;

System.out.println("pi1pi2D.test(101) " + pi1pi2D.test(101));
System.out.println("pi1pi2D.test(9) " + pi1pi2D.test(9));
System.out.println("pi1pi2D.test(11) " + pi1pi2D.test(11));
System.out.println("pi1pi2D.test(99) " + pi1pi2D.test(99));
/*Program to display names starts with 'P' by using Predicate*/

String[] names={"Praveen","Prasad","Varma","Phani","Kiran"};

Predicate<String> startsWithP= s -> s.charAt(0)=='P';

for(String name: names) {
    if(startsWithP.test(name)) {
        System.out.println(name);
    }
}
```

In Stream API, Predicate is used in

- ✓ Stream<T> filter(Predicate<? super T> predicate)
- ✓ boolean anyMatch(Predicate<? super T> predicate)
- ✓ boolean allMatch(Predicate<? super T> predicate)
- ✓ boolean noneMatch(Predicate<? super T> predicate)

b.Function

Function is similar to Predicate but it returns an object.

Function has 2 important methods i.e.. apply() and compose()

For example,

```
/* Function */
Function<String, Integer> f = s ->s.length();
System.out.println(f.apply("Praveen"));
System.out.println(f.apply("Prasad"));
Function<String,String> fr= s1->s1.replaceAll(" ","");
System.out.println(fr.apply("Praveen Oruganti"));
```

In Stream API, Functional is mostly used in

- ✓ <R> Stream<R> map(Function<? super T, ? extends R> mapper)
- ✓ IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.
- ✓ IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double
- ✓ <A> A[] toArray(IntFunction<A[]> generator)
- ✓ <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

c.Consumer

Consumer takes single input and return nothing i.e. void.

For example,

```
/* Consumer */
Consumer<String> c=s->System.out.println(s);
c.accept("Praveen");
c.accept("Prasad");

for(String name: names) {
    if(startsWithP.test(name)) {
        c.accept(name);
    }
}

/*consumer joining is also possible with andThen() */
```

In Stream API, Consumer is mostly used in

- ✓ Stream<T> peek(Consumer<? super T> action)
- ✓ void forEach(Consumer<? super T> action)
- ✓ void forEachOrdered(Consumer<? super T> action)

d.Supplier

Supplier takes no input and returns output.

For example,

```
/*Supplier */
Supplier<String> supplier= () -> "This is Praveen Oruganti";
System.out.println(supplier.get());
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());
```

In Stream API, Supplier is used in

- ✓ public static<T> Stream<T> generate(Supplier<T> s)
- ✓ <R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

3.Double Colon :: Method Reference

Java8 provides new feature **method reference** to call a single method of functional interface.

Method reference can be used to refer both static and non-static(instance) methods.

```
package com.praveen.java8;

@FunctionalInterface
interface MethodReference {
    void display();
}

public class StaticMethodReferenceDemo {

    static void display() {
        System.out.println("display");
    }

    public static void main(String args[]) {

        /* With Methodreference */
        MethodReference methodReference = StaticMethodReferenceDemo::display;
        methodReference.display();

        /* With Lambda */
        MethodReference methodReferenceLambda = () -> StaticMethodReferenceDemo.display();
        methodReferenceLambda.display();
    }
}
```

Rule1: Method names can be different.

Rule2: Parameter should be same in both methods.

Rule3: Method reference can be used in functional interface only.

Rule4: Return type of both methods can be different.

We can say MethodReference is alternative syntax for Lambda Expression.

```
package com.praveen.java8;

@FunctionalInterface
interface MethodReferenceNS {
    void display();
}

public class NonStaticMethodReferenceDemo {

    void display() {
        System.out.println("display");
    }

    public static void main(String args[]) {

        NonStaticMethodReferenceDemo obj = new NonStaticMethodReferenceDemo();

        /* With Methodreference */
        MethodReferenceNS methodReference = obj::display;
        methodReference.display();

        /* With Lambda */
        MethodReferenceNS methodReferenceLambda = () -> obj.display();
        methodReferenceLambda.display();
    }
}
```

For non-static method reference we just need to create an object and call the method whereas for static method reference you can use Class Name directly.

4.Optional Class

Optional is used for handling NullPointerException.

The class java.util.Optional is implemented as a single immutable concrete class that internally handles two cases; one with an element and one without.

Optional has different methods like of(),ofNullable(),isPresent(),ifPresent(),orElse(),.orElseThrow(),get()

For example,

```
/* Optional */
String[] words = new String[10];
Optional<String> checkNull =
              Optional.ofNullable(words[5]);

if (checkNull.isPresent()) {
    String word = words[5].toLowerCase();
    System.out.print(word);
} else {
    System.out.println("word is null");
}

words[2] = "This is Praveen Oruganti";
Optional<String> value = Optional.of(words[2]);
System.out.println(value.get());

System.out.println("Empty Optional: " + Optional.empty());

value.ifPresent(word -> {
    System.out.println("word name = " + word);
});
```

In Stream API, Optional is used in

- ✓ Optional<T> reduce(BinaryOperator<T> accumulator)
- ✓ Optional<T> min(Comparator<? super T> comparator)
- ✓ Optional<T> max(Comparator<? super T> comparator)
- ✓ Optional<T> findFirst()
- ✓ Optional<T> findAny()

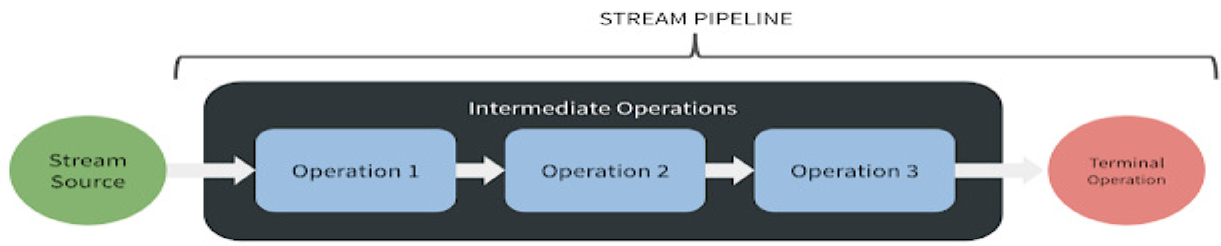
5.Stream API

Stream API is used to process the collection of objects.

Why we need Stream?
1. We can achieve Functional Programming
2. Code Reduce
3. Bulk operation
4. Parallel streams make it very easy for multithreaded operations

Stream Pipeline

A Stream pipeline consist of source, a zero or more intermediate operations and a terminal operation.

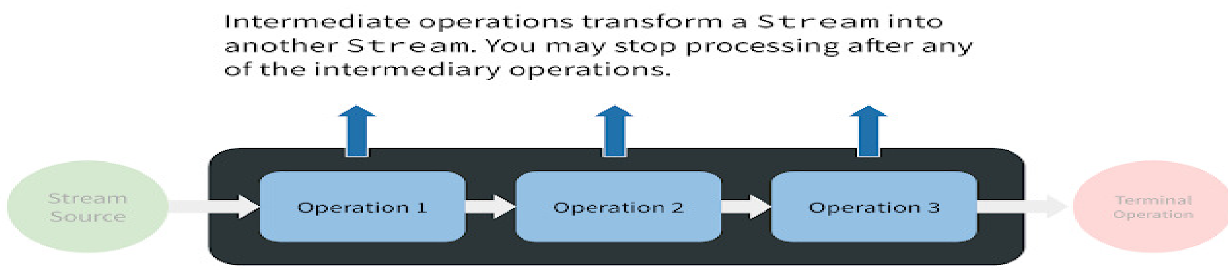


Stream Sources

Streams are mainly suited for handling collections of objects and can operate on elements of any type T. Although, there exist three special Stream implementations; IntStream, LongStream, and DoubleStream which are restricted to handle the corresponding primitive types.
An empty Stream of any of these types can be generated by calling Stream.empty() in the following manner:
Stream<T> Stream.empty()
IntStream IntStream.empty()
LongStream LongStream.empty()
DoubleStream DoubleStream.empty()

Intermediate operations

Intermediate operations act as a declarative (functional) description of how elements of the Stream should be transformed. Together, they form a pipeline through which the elements will flow. What comes out at the end of the line, naturally depends on how the pipeline is designed.



- ✓ filter(Predicate<T>)
- ✓ map(Function<T>)
- ✓ flatmap(Function<T>)
- ✓ sorted(Comparator<T>)
- ✓ peek(Consumer<T>)
- ✓ distinct()
- ✓ limit(long n)
- ✓ skip(long n)

```
List<String> list = Stream.of("Monkey", "Lion", "Giraffe","Lemur")
    .filter(s -> s.startsWith("L"))
    .map(String::toUpperCase)
    .sorted()
    .collect(toList());

System.out.println(list);

 [LEMUR, LION]
```

Stream filter()

filter allows you to filter the stream to process the stream. **Syntax** : Stream<T> filter(Predicate<? super T> predicate)

For example,

```
eList.stream().filter(e -> e.getEmpLocation().equalsIgnoreCase("Hyderabad")).forEach(System.out::println);
```

How filter works internally?

filter method takes a predicate as an input which is an functional interface. Java will convert the above filter syntax into below lines of code by adding a Predicate functional interface.

Java converts the above filter code by adding a Predicate class and overrides the test method as shown below:

```
eList.stream().filter(new Predicate<Employee1>() {
    public boolean test(Employee1 e) {
        return "Hyderabad".equalsIgnoreCase(e.getEmpLocation());
    }
}).forEach(System.out::println);
```

usage of filter(),sort() and findFirst() methods

```
System.out.println(eList.stream().filter(e -> e.getEmpLocation().equalsIgnoreCase("Hyderabad"))
        .sorted((Employee1 emp1, Employee1 emp2)-> emp1.getEmpName().compareTo(emp2.getEmpName())).findFirst().get());
```

Usage if filter() and allMatch() methods

My requirement is I have a map and a list. The map contains ID and an Employee object. List contains employee names. I want to filter out the names from employee list which are not present in the map.

```
Map<String, Employee1> empMap = new HashMap<>();
empMap.put("1", new Employee1( "Praveen",149903L,34,"Hyderabad"));
empMap.put("2", new Employee1( "Prasad",149904L,35,"Hyderabad"));
empMap.put("3", new Employee1( "Varma",149905L,36,"Bangalore"));

List<String> empNames=Arrays.asList("Sharma","Praveen","Varma","Krishna");

List<String> filteredList=empNames.stream()
                                .filter(emp->empMap.entrySet().stream().allMatch(entry->!entry.getValue().getEmpName().equals(emp)))
                                .collect(Collectors.toList());
System.out.println(filteredList);
```

map()

<R> Stream<R> map(Function<? super T, ? extends R> mapper)

One of the most versatile operations we can apply to a Stream is map(). It allows elements of a Stream to be transformed into something else by mapping them to another value or type. This means the result of this operation can be a Stream of any type R. The example below performs a simple mapping from String to String, replacing any capital letters with their lower case equivalent.

Suppose we want to get the age of the Employee whose name is Praveen

```
System.out.println(eList.stream().filter(e -> e.getEmpName().equalsIgnoreCase("praveen"))
        .map(Employee1::getEmpAge).findAny().orElse(0));
```

```
Stream<String> lowerCase = Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur"
)
   .map(String::toLowerCase);

lowerCase: [monkey, lion, giraffe, lemur]

IntStream lengths = Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur"
)
   .mapToInt(String::length);

lengths: [6, 4, 7, 5]
```

Usage of mapToInt and collect as a List in Java8

```
List<Integer> numList = Arrays.asList(1, 9, 8, 5);
System.out.println(numList.stream().
mapToInt(num->num*5).collect(ArrayList::new, ArrayList::add,
ArrayList::addAll));
```

(or) we can also code like below

```
System.out.println(numList.stream().mapToInt(num-
>num*5).boxed().collect(Collectors.toList()));
```

Output:

[5, 45, 40, 25]

flatMap()

The last operation that we will cover in this article might be more tricky to understand even though it can be quite powerful. It is related to the map() operation but instead of taking a Function that goes from a type T to a return type R, it takes a Function that goes from a type T and returns a Stream of R. These “internal” streams are then flattened out to the resulting streams resulting in a concatenation of all the elements of the internal streams.

```
Stream<Character> chars = Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur"
)
    .flatMap(s -> s.chars().mapToObj(i -> (char) i));

chars: [M, o, n, k, e, y, L, i, o, n, G, i, r, a, f, f, e, L, e, m, u, r]
```

map() vs flatMap()

map() is used to transform one Stream into another by applying a function on each element and flatMap() does both transformation as well as flattening. The flatMap() function can take a Stream of List and return Stream of values combined from all those list.

```
List<List<Integer>> listOfListOfNumber = new ArrayList<>();
listOfListOfNumber.add(Arrays.asList(2, 4));
listOfListOfNumber.add(Arrays.asList(3, 9));
listOfListOfNumber.add(Arrays.asList(4, 16));
System.out.println(listOfListOfNumber);
System.out.println(listOfListOfNumber.stream().flatMap(list -> list.stream()).collect(Collectors.toList()));
```

Output

```
[[2, 4], [3, 9], [4, 16]]
[2, 4, 3, 9, 4, 16]
```

reduce

The reduction operation combines all elements of the stream into a single result. Java 8 supports three different kind of reduce methods.

The **first** one reduces a stream of elements to exactly one element of the stream. Let's see how we can use this method to determine the oldest Employee:

```
eList.stream().reduce((emp1, emp2) -> emp1.getEmpAge() > emp2.getEmpAge() ? e1 : e2)
        .ifPresent(System.out::println);
```

The **second** reduce method accepts both an identity value and a BinaryOperator accumulator. This method can be utilized to construct a new Person with the aggregated names and ages from all other persons in the stream:

```
Employee1 result = eList.stream().reduce(new Employee1("", 0L, 0, ""), (emp1, emp2) -> {
    emp1.empAge += emp2.empAge;
    emp1.empName += emp2.empName;
    return emp1;
});
System.out.format("name=%s; age=%s\n", result.empName, result.empAge);
```

The **third** reduce method accepts three parameters: an identity value, a BiFunction accumulator and a combiner function of type BinaryOperator. Since the identity values type is not restricted to the Person type, we can utilize this reduction to determine the sum of ages from all employees:

```
Integer ageSum = eList
        .stream()
        .reduce(0, (sum, emp) -> sum += emp.getEmpAge(), (sum1, sum2) -> sum1 + sum2);

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

```
public long factorial(int n) {
    if (n > 20) throw new IllegalArgumentException(n + " is out of range");
    return LongStream.rangeClosed(2, n).reduce(1, (a, b) -> a * b);
}
```

distinct()

```java
// For example, i have an array list which has duplicate elements.
List<Integer> intList=Arrays.asList(1,2,3,2,6,7,5,3,1);
System.out.println(intList); // [1, 2, 3, 2, 6, 7, 5, 3, 1]
// WAP to Remove the duplicates and print the list using stream api
System.out.println(intList.stream()
                          .distinct()
                          .collect(Collectors.toList())); // [1, 2, 3, 6, 7, 5]
// WAP to Print the duplicated list using stream api.
System.out.println(intList.stream()
                          .filter(i -> Collections.frequency(intList, i) >1)
                          .collect(Collectors.toSet())); // [1, 2, 3]
// WAP to Print the non duplicated list using stream api.
System.out.println(intList.stream()
                          .filter(i -> Collections.frequency(intList, i) ==1)
                          .collect(Collectors.toList())); // [6, 7, 5]
//  WAP to Print the frequency of elements present in the given list using stream api.
System.out.println(intList.stream().
                           collect(Collectors.groupingBy(Function.identity(),Collectors.counting()))); // {1=2, 2=2, 3=2, 5=1, 6=1, 7=1}
```

Terminal operations

Now that we are familiar with the initiation and construction of a Stream pipeline we need a way to handle the output. Terminal operations allow this by producing a result from the remaining elements (such as count()) or a side-effect (such as forEach(Consumer)).

A Stream will not perform any computations on the elements of the source before the terminal operation is initiated. This means that source elements are consumed only as needed - a smart way to avoid unnecessary work. This also means that once the terminal operation is applied, the Stream is consumed and no further operations can be added.

- ✓ forEach
- ✓ ForEachOrdered
- ✓ toArray
- ✓ reduce T reduce(T identity, BinaryOperator<T> accumulator);
- ✓ collect
- ✓ min
- ✓ max
- ✓ count
- ✓ anyMatch
- ✓ allMatch
- ✓ noneMatch

- ✓ findFirst
- ✓ findAny

```
Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
    .forEachOrdered(System.out::print);
```

This will produce the following output:

```
MonkeyLionGiraffeLemurLion
```

Occurrence of Elements

The intermediate operation filter() is a great way to eliminate elements that do not match a given predicate. Although, in some cases, we just want to know if there is at least one element that fulfills the predicate. If so, it is more convenient and efficient to use anyMatch(). Here we look for the occurrence of the number 2:

```
boolean containsTwo = IntStream.of(1, 2, 3).anyMatch(i -> i == 2);
```

```
containsTwo: true
```

Operations for Calculation

Several terminal operations output the result of a calculation. The simplest calculation we can perform being count() which can be applied to any Stream. It can, for example, be used to count the number of animals:

```
long nrOfAnimals = Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur"
)
 .count();
```

```
nrOfAnimals: 4
```

Although, some terminal operations are only available for the special Stream implementations; IntStream, LongStream and DoubleStream. Having access to a Stream of such type we can simply sum all the elements like this:

```
int sum = IntStream.of(1, 2, 3).sum();
```

```
sum: 6
```

Or why not compute the average value of the integers with .average():

```
OptionalDouble average = IntStream.of(1, 2, 3).average();
```

```
average: OptionalDouble[2.0]
```

Or retrieve the maximal value with .max().

```
int max = IntStream.of(1, 2, 3).max().orElse(0);
```

```
max: 3
```

Like average(), the result of the max() operator is an Optional, hence by stating .orElse(0) we automatically retrieve the value if its present or fall back to 0 as our default. The same solution can be applied to the average-example if we rather deal with a primitive return type.

summaryStatistics()

It is used to calculate the sum, min, max, avg and total count of the elements passed to this method

summarizingInt(), summarizingLong(), summarizingDouble()

These methods return a special class called Int/Long/ DoubleSummaryStatistics which contain statistical information like sum, max, min, average etc of input elements.

```
IntSummaryStatistics statistics = eList.stream().mapToInt(Employee1::getEmpAge).summaryStatistics();

System.out.println(statistics.getCount());
System.out.println(statistics.getSum());
System.out.println(statistics.getMin());
System.out.println(statistics.getMax());
System.out.println(statistics.getAverage());
```

```
IntSummaryStatistics statistics = IntStream.of(1, 2, 3).summaryStatistics();
```

```
statistics: IntSummaryStatistics{count=3, sum=6, min=1, average=2.000000, max=3}
```

Collectors

Collectors.toList()	*Collectors.toSet()*	*Collectors.toMap()*	*Collectors.toCollection()*
Collectors.joining()	*Collectors.counting()*	*Collectors.collectingAndThen()*	
Collectors.maxBy()	*Collectors.minBy()*		
Collectors.summingInt()	*Collectors.summingLong()*	*Collectors.summingDouble()*	
Collectors.groupingBy()	*Collectors.partitioningBy()*		
Collectors.averagingInt()	*Collectors.averagingLong()*	*Collectors.averagingDouble()*	
Collectors.summarizingInt()	*Collectors.summarizingLong(*	*Collectors.summarizingDouble()*	

1.**toSet()**

We can collect all elements into a Set simply by collecting the elements of the Stream with the collector toSet().

```
Set<String> collectToSet = Stream.of(
   "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
   .collect(Collectors.toSet());
```

```
toSet: [Monkey, Lion, Giraffe, Lemur]
```

2. **toList()**

Similarly, the elements can be collected into a List using toList() collector.

```
List<String> collectToList = Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
    .collect(Collectors.toList());


collectToList: [Monkey, Lion, Giraffe, Lemur, Lion]
```

3. **toCollection()**

In a more general case, it is possible to collect the elements of the Stream into any Collection by just providing a constructor to the desired Collection type. Example of constructors are LinkedList::new, LinkedHashSet::new and PriorityQueue::new

```
LinkedList<String> collectToCollection = Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
    .collect(Collectors.toCollection(LinkedList::new));


collectToCollection: [Monkey, Lion, Giraffe, Lemur, Lion]
String[] toArray = Stream.of(
   "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
   .toArray(String[]::new);


toArray: [Monkey, Lion, Giraffe, Lemur, Lion]
Map<String, Integer> toMap = Stream.of(
    "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
   .distinct()
   .collect(Collectors.toMap(
       Function.identity(),   //Function<String, K> keyMapper
       s -> (int) s.chars().distinct().count()// Function<String, V> valueMapper
   ));


toMap: {Monkey=6, Lion=4, Lemur=5, Giraffe=6}   (*)
```

4. **joining()**

Collectors.joining will join all the results with delimiter specified in the parameter.

```
List<String> fruitList = Arrays.asList("banana", "apple", "mango", "grapes");
System.out.println(fruitList.stream().map(Function.identity()).collect(Collectors.joining(" | ")));

System.out.println(eList.stream().map(Employee1::getEmpName).collect(Collectors.joining("|")));
```

Output

banana | apple | mango | grapes
Praveen|Khaja|Varma|Hari|Krishna

5.**partitioningBy()**
We can partition a set of stream in two based on certain condition. So it will create two streams – one which satisfies the condition and another which does not satisfies the conditions.
partitioningBy will return a map containing two keys
true – which holds the stream which satisfy the condition.
false - which holds the stream which does not satisfy the condition.

```
Map<Boolean, List<Employee1>> partition = eList.stream()
        .collect(Collectors.partitioningBy(e -> e.getEmpLocation().equals("Hyderabad")));
System.out.println("Employee1s working in Hyderabad Location " + partition.get(true));
System.out.println("Employee1s working in other Location " + partition.get(false));
```

Employee1s working in Hyderabad Location [Employee1 [empName=Praveen, empId=149903, empAge=34, empLocation=Hyderabad], Employee1 [empName=Hari, empId=89778, empAge=43, empLocation=Hyderabad]]
Employee1s working in other Location [Employee1 [empName=Khaja, empId=250005, empAge=35, empLocation=Bangalore], Employee1 [empName=Varma, empId=26767, empAge=36, empLocation=Singapore], Employee1 [empName=Krishna, empId=22203, empAge=38, empLocation=SouthAfrica]]

6. **groupingBy()**
groupingBy() is used to group the stream based on the condition passed to the groupingBy()

```
String sentence = "mango apple mango banana apple mango apple banana mango";
String[] words = sentence.split(" ");
System.out.println(Arrays.asList(words).stream()
        .collect(Collectors.groupingBy(Function.identity(), Collectors.counting())));
```

Output
{banana=2, apple=3, mango=4}

```
Map<String, List<Employee1>> groupBy = eList.stream().collect(Collectors.groupingBy(Employee1::getEmpLocation));
System.out.println(groupBy);
```

Output
{Singapore=[Employee1 [empName=Varma, empId=26767, empAge=36, empLocation=Singapore]], SouthAfrica=[Employee1 [empName=Krishna, empId=22203, empAge=38, empLocation=SouthAfrica]], Hyderabad=[Employee1 [empName=Praveen, empId=149903, empAge=34, empLocation=Hyderabad], Employee1 [empName=Hari, empId=89778, empAge=43, empLocation=Hyderabad]], Bangalore=[Employee1 [empName=Khaja, empId=250005, empAge=35, empLocation=Bangalore]]}

```
Map<Character, List<String>> groupingByList =  Stream.of(
     "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
    .collect(Collectors.groupingBy(
        s -> s.charAt(0) // Function<String, K> classifier
    ));
```

```
groupingByList: {G=[Giraffe], L=[Lion, Lemur, Lion], M=[Monkey]}
```

```
Map<Character, Long> groupingByCounting =  Stream.of(
     "Monkey", "Lion", "Giraffe", "Lemur", "Lion"
)
    .collect(Collectors.groupingBy(
        s -> s.charAt(0), // Function<String, K> classifier
        counting()        // Downstream collector
    ));
```

```
groupingByCounting: {G=1, L=3, M=1}
```

7. **mappingBy**

mappingBy() allows us to pick the particular property of the Object to store into map rather than storing the complete Object

```
Map<String, Set<String>> mappingBy = eList.stream().collect(Collectors.groupingBy(Employee1::getEmpLocation,
        Collectors.mapping(Employee1::getEmpName, Collectors.toSet())));
System.out.println(mappingBy);
```

Output

{Singapore=[Varma], SouthAfrica=[Krishna], Hyderabad=[Hari, Praveen], Bangalore=[Khaja]}

8. **counting()**

It returns a Collector that counts number of input elements.

9. **maxBy()**

This method returns a Collector that collects largest element in a stream according to supplied Comparator.

```
Map<String, Employee1> empMap = new HashMap<>();
empMap.put("1", new Employee1( "Praveen",149903L,34,"Hyderabad"));
empMap.put("2", new Employee1( "Prasad",149904L,35,"Hyderabad"));
empMap.put("3", new Employee1( "Varma",149905L,36,"Bangalore"));

System.out.println(empMap.entrySet().stream().map(emp->emp.getValue().empAge).collect(Collectors.maxBy(Comparator.naturalOrder())));
```

10. **minBy()**

This method returns a Collector which collects smallest element in a stream according to supplied Comparator.

```
Map<String, Employee1> empMap = new HashMap<>();
empMap.put("1", new Employee1( "Praveen",149903L,34,"Hyderabad"));
empMap.put("2", new Employee1( "Prasad",149904L,35,"Hyderabad"));
empMap.put("3", new Employee1( "Varma",149905L,36,"Bangalore"));

System.out.println(empMap.entrySet().stream().map(emp->emp.getValue().empAge).collect(Collectors.minBy(Comparator.naturalOrder())));
```

11. **summingInt(), summingLong(), summingDouble()**
These methods returns a Collector which collects sum of all input elements.
12. **averagingInt(), averagingLong(), averagingDouble()**
These methods return a Collector which collects average of input elements.
13. **summarizingInt(), summarizingLong(), summarizingDouble()**
These methods return a special class called Int/Long/ DoubleSummaryStatistics which contain statistical information like sum, max, min, average etc of input elements.

Parallel Streams

Streams can be executed in parallel to increase runtime performance on large amount of input elements. Parallel streams use a common ForkJoinPool available via the static ForkJoinPool.commonPool() method. The size of the underlying thread-pool uses up to five threads - depending on the amount of available physical CPU cores:

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());    // 3
```

On my machine the common pool is initialized with a parallelism of 3 per default. This value can be decreased or increased by setting the following JVM parameter:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Collections support the method parallelStream() to create a parallel stream of elements. Alternatively you can call the intermediate method parallel() on a given stream to convert a sequential stream to a parallel counterpart.

In order to understate the parallel execution behavior of a parallel stream the next example prints information about the current thread to sout:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
.parallelStream()
.filter(s -> {
    System.out.format("filter: %s [%s]\n",
        s, Thread.currentThread().getName());
    return true;
})
.map(s -> {
    System.out.format("map: %s [%s]\n",
        s, Thread.currentThread().getName());
    return s.toUpperCase();
})
.forEach(s -> System.out.format("forEach: %s [%s]\n",
    s, Thread.currentThread().getName()));
```

By investigating the debug output we should get a better understanding which threads are actually used to execute the stream operations:

```
filter: b1 [main]
map: b1 [main]
forEach: B1 [main]
filter: a2 [ForkJoinPool.commonPool-worker-1]
map: a2 [ForkJoinPool.commonPool-worker-1]
filter: c1 [main]
map: c1 [main]
forEach: C1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
filter: c2 [ForkJoinPool.commonPool-worker-2]
map: c2 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-2]
filter: a1 [ForkJoinPool.commonPool-worker-3]
map: a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
```

As you can see the parallel stream utilizes all available threads from the common ForkJoinPool for executing the stream operations. The output may differ in consecutive runs because the behavior which particular thread is actually used is non-deterministic.

Let's extend the example by an additional stream operation, sort:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
.parallelStream()
.filter(s -> {
    System.out.format("filter: %s [%s]\n",
        s, Thread.currentThread().getName());
    return true;
})
.map(s -> {
    System.out.format("map: %s [%s]\n",
        s, Thread.currentThread().getName());
    return s.toUpperCase();
})
.sorted((s1, s2) -> {
    System.out.format("sort: %s <> %s [%s]\n",
        s1, s2, Thread.currentThread().getName());
    return s1.compareTo(s2);
})
.forEach(s -> System.out.format("forEach: %s [%s]\n",
    s, Thread.currentThread().getName()));
```

The result may look strange at first:

```
filter: b1 [main]
filter: a1 [ForkJoinPool.commonPool-worker-2]
map: a1 [ForkJoinPool.commonPool-worker-2]
filter: c1 [ForkJoinPool.commonPool-worker-3]
filter: a2 [ForkJoinPool.commonPool-worker-1]
map: c1 [ForkJoinPool.commonPool-worker-3]
filter: c2 [ForkJoinPool.commonPool-worker-2]
map: b1 [main]
map: c2 [ForkJoinPool.commonPool-worker-2]
map: a2 [ForkJoinPool.commonPool-worker-1]
sort: A2 <> A1 [main]
sort: B1 <> A2 [main]
sort: C2 <> B1 [main]
sort: C1 <> C2 [main]
sort: C1 <> B1 [main]
sort: C1 <> C2 [main]
forEach: B1 [main]
forEach: C1 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
```

It seems that sort is executed sequentially on the main thread only. Actually, sort on a parallel stream uses the new Java 8 method Arrays.parallelSort() under the hood.

6. New Date-Time API

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :

1. Not thread safe : Unlike old java.util.Date which is not thread safe the new date-time API is immutable and doesn't have setter methods.

2. Less operations : In old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

1. Local : Simplified date-time API with no complexity of timezone handling.
2. Zoned : Specialized date-time API to deal with various timezones.

1. LocalDate/LocalTime: LocalDate and LocalTime Classes is introduce where timezones are not required.

2. Zone DateTime API: It is used when time zone is to be

3.ChronoUnit Enum: ChronoUnit enum is added in the new Java 8 API which is Used to represent day, month, etc and it is available in java.time.temporal package.

For Example,

```
public class NewDateTimeApiExamples {

    public static void main(String args[]) {

        // current date and time
        LocalDateTime currentTime = LocalDateTime.now();
        System.out.println("Current DateTime: " + currentTime);

        LocalDate date1 = currentTime.toLocalDate();
        System.out.println("date1: " + date1);

        Month month = currentTime.getMonth();
        int day = currentTime.getDayOfMonth();
        int seconds = currentTime.getSecond();

        System.out.println("Month: " + month + " day: " + day + " seconds: " + seconds);

        // current date and time
        ZonedDateTime date = ZonedDateTime.parse("2019-07-28T10:14:20+06:30[Asia/Kolkata]");
        System.out.println("date: " + date);

        // Get the current date
        LocalDate currentDate = LocalDate.now();
        System.out.println("Current date: " + currentDate);

        // add 1 week to the current date
        LocalDate nextWeek = currentDate.plus(1, ChronoUnit.WEEKS);
        System.out.println("Next week: " + nextWeek);
```

```java
        // add 2 month to the current date
        LocalDate nextMonth = currentDate.plus(2, ChronoUnit.MONTHS);
        System.out.println("Next month: " + nextMonth);

        // add 3 year to the current date
        LocalDate nextYear = currentDate.plus(3, ChronoUnit.YEARS);
        System.out.println("Next year: " + nextYear);

        // add 10 years to the current date
        LocalDate nextDecade = currentDate.plus(1, ChronoUnit.DECADES);
        System.out.println("Next ten year: " + nextDecade);

        // comparing dates
        LocalDate date2 = LocalDate.of(2014, 1, 15);
        LocalDate date3 = LocalDate.of(2019, 7, 28);

        if (date2.isAfter(date3)) {
            System.out.println("date2 comes after date3");
        } else {
            System.out.println("date2 comes before date3");
        }

        // check Leap year
        if (date1.isLeapYear()) {
            System.out.println("This year is Leap year");
        } else {
            System.out.println(date1.getYear() + " is not a Leap year");
        }

        //How many days, month between two dates
        LocalDate newDate = LocalDate.of(2019, Month.DECEMBER, 14);
        Period periodTonewDate = Period.between(date1, newDate);
        System.out.println("Months left between today and newDate : " + periodTonewDate.getMonths());
```

Output

```
Current DateTime: 2019-10-27T13:03:43.070
date1: 2019-10-27
Month: OCTOBER day: 27 seconds: 43
date: 2019-07-28T10:14:20+05:30[Asia/Kolkata]
Current date: 2019-10-27
Next week: 2019-11-03
Next month: 2019-12-27
Next year: 2022-10-27
Next ten year: 2029-10-27
date2 comes before date3
2019 is not a Leap year
Months left between today and newDate : 1
```

Working with Files

The utility class Files was first introduced in Java 7 as part of Java NIO. The JDK 8 API adds a couple of additional methods which enables us to use functional streams with files.

Listing files

The method Files.list streams all paths for a given directory, so we can use stream operations like filter and sorted upon the contents of the file system.

```java
// Listing files
try (Stream<Path> stream = Files.list(Paths.get(""))) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> !path.startsWith("."))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("List: " + joined);
}
```

The above example lists all files for the current working directory, then maps each path to its string representation. The result is then filtered, sorted and finally joined into a string.

Finding files

```
// Finding files
Path start = Paths.get("src/main/java");
int maxDepth = 5;
try (Stream<Path> stream = Files.find(start, maxDepth, (path, attr) ->
        String.valueOf(path).endsWith(".java"))) {
    String joined = stream
        .sorted()
        .map(String::valueOf)
        .collect(Collectors.joining("; "));
    System.out.println("Found: " + joined);
}
```

The method find accepts three arguments: The directory path start is the initial starting point and maxDepth defines the maximum folder depth to be searched. The third argument is a matching predicate and defines the search logic. In the above example we search for all Java files (filename ends with .java).

We can achieve the same behavior by utilizing the method Files.walk. Instead of passing a search predicate this method just walks over any file.

```
// Finding files
Path start1 = Paths.get("src/main/java");
int maxDepth1 = 5;
try (Stream<Path> stream = Files.walk(start1, maxDepth1)) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> path.endsWith(".java"))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("walk(): " + joined);
}
```

In this example we use the stream operation filter to achieve the same behavior as in the previous example.

Reading and writing files

Reading text files into memory and writing strings into a text file in Java 8 is finally a simple task. No messing around with readers and writers. The method Files.readAllLines reads all lines of a given file into a list of strings. You can simply modify this list and write the lines into another file via Files.write:

```java
// Reading file
List<String> lines = Files.readAllLines(Paths.get("file.txt"));
System.out.println(lines.size());
// Writing file
lines.add("How are you?");
Files.write(Paths.get("file.txt"), lines);
```

Please keep in mind that those methods are not very memory-efficient because the whole file will be read into memory. The larger the file the more heap-size will be used.

As an memory-efficient alternative you could use the method Files.lines. Instead of reading all lines into memory at once, this method reads and streams each line one by one via functional streams.

```java
// Reading file using streams
try (Stream<String> stream = Files.lines(Paths.get("file.txt"))) {
    stream
        .filter(line -> line.contains("Praveen"))
        .map(String::trim)
        .forEach(System.out::println);
}
```

If you need more fine-grained control you can instead construct a new buffered reader (or) in case you want to write to a file simply construct a buffered writer instead:

```java
Path path = Paths.get("file1.txt");
try (BufferedWriter writer = Files.newBufferedWriter(path)) {
    writer.write("How are you?");
}
try (BufferedReader reader = Files.newBufferedReader(path)) {
    System.out.println(reader.readLine());
}
```

Buffered readers also have access to functional streams. The method lines construct a functional stream upon all lines denoted by the buffered reader:

```java
Path path = Paths.get("file1.txt");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    long countPrints = reader
        .lines()
        .filter(line -> line.contains("you"))
        .count();
    System.out.println(countPrints);
}
```

So as you can see Java 8 provides three simple ways to read the lines of a text file, making text file handling quite convenient.

Unfortunately you have to close functional file streams explicitly with try/with statements which makes the code samples still kind of cluttered. I would have expected that functional streams auto-close when calling a terminal operation like count or collect since you cannot call terminal operations twice on the same stream anyway.

You can find complete code of Java 8 Features in my repository (https://github.com/praveenoruganti/praveen-java8-examples)