**Method References**

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method,

**Optional Class**

To avoid abnormal termination, we use the Optional class. A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

**Default Method**

Before Java 8, interfaces could have only abstract methods. 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.

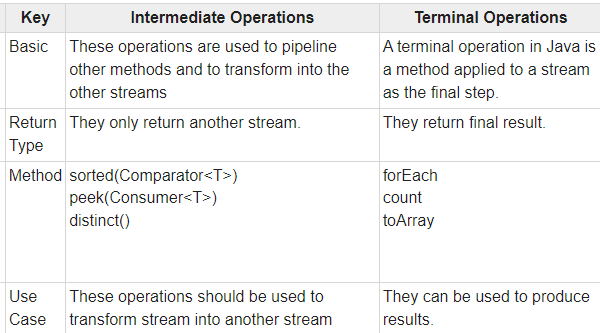
**Streams**

Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

A **Collection**is an in-memory data structure, which holds all the values that the data structure currently has. Every element in the Collection has to be computed before we add it to the Collection. Operations such as searching, sorting, insertion, manipulation, and deletion can be performed on a Collection.

 A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

**Difference Between Intermediate And Terminary Operations**



**Pipelining in java**

Stream pipelining is the concept of chaining operations together. This is done by splitting the operations that can happen on a stream into two categories. They are "intermediate operations" and "terminal operations". Each intermediate operation returns an instance of Stream itself when it runs, an arbitrary number of intermediate operations can, therefore, be set up to process data forming a processing pipeline. There must then be a terminal operation which returns a final value and terminates the pipeline.

**Filter Method In java**

returns a stream consisting of the elements of this stream that match the given predicate. This is an ***intermediate operation.*** These operations are always lazy i.e, executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.

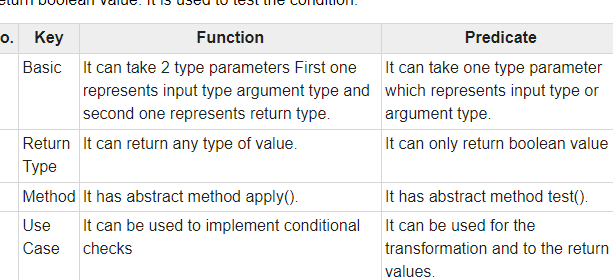
**Convert Array To Stream**

The **stream(T[] array)** method of **Arrays class in Java**, is used to get a Sequential Stream from the array passed as the parameter with its elements. It returns a sequential Stream with the elements of the array, passed as parameter, as its source.

**Syntax:**

public static <T> Stream<T> stream(T[] array)

**Difference Between Function And Predicate**



**Java 8 date and Time api**

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.

* **LocalDate/LocatTime**and **LocalDateTime API :**Use it when time zones are NOT required.

Zoned date-time API : Use it when time zones are to be considered

* **Period**and **Duration** classes :   
  *Period :*It deals with *date* based amount of time.   
  *Duration :* It deals with*time*based amount of time.

ChronoUnits Enum : java.time.temporal.ChronoUnit enum is added in Java 8 to replace integer values used in old API to represent day, month etc.

**TemporalAdjuster :**It is used to perform various date related operations

**How to get Current date and time**

**LocalDateTime** class which gives both date and time part. Use **now()** method to **get the current date and time**

Next java 8 class is **ZonedDateTime** and this works with timezone part along with the date time.Use **now()** method to get the current date time from the current timezone.

**Custom Functional Interface**

 We can declare our **own/custom** **functional interface**by defining the**Single Abstract Method (SAM)** in an interface.

**FunctionalInterface Annotation**

@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message.

**Consumer Functional Interface**

Represents an operation that accepts a single input argument and returns no result

method is accept(Object)..

        // Calling Predicate method

        System.out.println(lesserthan.test(10));

**Function**

 It represents a function which takes in one argument and produces a result. Hence this functional interface takes in 2 generics namely as follows:

apply()

andThen()

compose()

identity()

  Function<Integer, Double> half = a -> a / 2.0;

        // Applying the function to get the result

        System.out.println(half.apply(10));