A **Stream is a conceptually fixed data structure, in which elements are computed on demand**. “Like Video Stream Concept i.e video will play as soon as 1% of video is downloaded and Streaming of remaining video will happen in backend”.

Stream **operations are either**

1. **Intermediate or**
2. **Terminal**.

While **terminal operations return a result of a certain type**, **intermediate operations return the stream itself**

Streams help us to iterate and do operations on Collections API efficiently.

1. Stream reduces developer coding cycle.
2. Stream operation can be executed either serial or parallel.

**Ex Scenario:** In “List<Employee> personList” get average age of male employees. Lets say there are 1000 employees.

**Traditional Approach:** We will do for loop. Get employees one by one, check for male, aggregate the result. Then finally take the count of total employees. Divide the aggregate result to count.

**Stream Approach1:**

double average = roster

.stream()

.filter(p -> p.getGender() == Person.Sex.MALE)

.mapToInt(Person::getAge)

.average()

.getAsDouble();

**Stream Approach2:** Here in below approach, using ForkJoinTask/ForkJoinPool concept problem is divided into subproblems(divide and conquer) i.e 1000 employees are split into 100 and operation is paralleled.

double average = roster

.parallelStream()

.filter(p -> p.getGender() == Person.Sex.MALE)

.mapToInt(Person::getAge)

.average()

.getAsDouble();

List<Integer> transactionsIds =

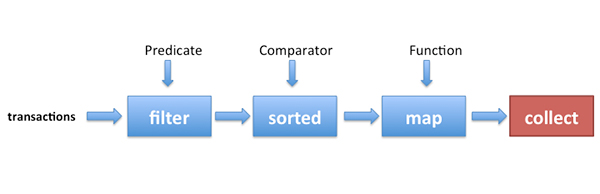
transactions.stream()

.filter(t -> t.getType() == Transaction.GROCERY)

.sorted(comparing(Transaction::getValue).reversed())

.map(Transaction::getId)

.collect(toList());



List<Integer> transactionsIds =

transactions.parallelStream()

.filter(t -> t.getType() == Transaction.GROCERY)

.sorted(comparing(Transaction::getValue).reversed())

.map(Transaction::getId)

.collect(toList());

