1. **Definition**

* Lambda is **anonymous function – a method without a name.**
* It consists of 3 components -> **method params, method body, return type**
* There are 4 primary interfaces: Consumer, Predicate, Function, Supplier
* Syntax: (params) -> {lambda body}

1. **Usage**

* Mainly used to implement **Functional Interfaces** (interface having only 1 abstract method)**.**

1. **Category**

* Consumer Interface:

+ As a consumer, this interface has an abstract method **taking 1 / 2 input, perform some action and return nothing**.

+ Function: void accept (T param);

+ default **chaining method** is ‘andThen(Consumer)’.

+ forEach function is taking consumer interface as follow:

Ex: List.forEach(s -> print(s.toUpperCase())

* Predicate Interface

+ As a predicate, this interface has an abstract method **taking 1 /2 inputs, perform some action and return Boolean value.**

**+** Function: Boolean test (T param)’

+ default **chaining methods** are ‘and(Predicate)’ vs ‘or(Predicate)’ vs negate().

* Function Interface

+ A function interface has 1 abstract method **taking 1 / 2 inputs, perform some action and return a Class Type value**.

+ Function: R apply(T t);

+ default **chaining method** is ‘andThen(Function)’.

+ Note that if the input(s) type is the same type with the output, UnaryOperator and BinaryOperator is a good alternative solution.

* Suppler Interface

+ A supplier is an opposite interface compare to consumer interface. This interface take 0 input and return a class Object.

+ Function: T get();

**Stream API**

1. Definition

* Stream API is introduced in Java 8 and the main purpose of it is to perform some **Operation on Collections (sequentially or parallel).**
* Steam is a sequence of elements which can be created out from a Collection interfaces (Set / List / Queue), I / O resources and etc.

**Parallel Stream – NQ Model**

* NQ model consists of 2 factors which are N and Q:

+ N = number of data items

+ Q = amount of work per item

* The **rule of thumb** stating that in general, for parallelism to have a high chance of speeding up processes, **NQ > 10.000**.
* For cases that NQ < 10.000, a careful usage of parallelism is a must. If it does prove to be faster (due to some other factors – such as low locality, …) and you are facing with performance issue => apply it.
* There are still some other factors effecting the parallelism such as

+ **Split cost** (If the split cost is too high => time took for stream to split up data and dispatching it to multiple cores is terribly slow)

+ **Sub results combination costs – merging cost**. (If the result combination cost is excessively high, dispatching data to multiple cores and merging them back will significantly slow down your process)

+ **Locality of the data** (If the data is not well stored, fetching of the data and feeding it to the parallel processes will successively cause long idle times. Generally speaking, summing up adjacent (primitive / object) elements in storage is faster than (primitive / object) references – such as summing int[] vs summing Integer[])

+ **Encounter Order**

---- in details:

* **Split cost** is a factor that can be picked up at the first place so make sure to use the right data source:

+ Arrays split cheaply, evenly and with perfect knowledge of split sizes (Arrays split from element 0 to 100, 101 to 200, …)

+ Linked lists have none of the above properties so as a result, it causes the split cost to be immensely high. (linked lists store the addresses of its elements so when come to splitting, data get split to first elements and the rest elements, and then continue from the rest to the first rest and the last rest of the rest and eventually, it ends up to a heavy branches)

+ Hash map and trees are in the middle, in the term of splitting cost as they split reasonably balance but they don’t have good locality.

+ **Iterative generators** is on high split cost as it has to get the first iteration done to perform the next iteration step.

Ex: IntStream,iterate(0, i -> i + 1).limit(n).sum();

+ **Stateless generators** like Arrays, it has low cost in splitting things

Ex: IntStream.range(0, n).sum();

* **Encounter Order** is also one of a factor to be considered in the first place before applying parallelism as:

+ Some data sources have not defined encounter order like **Set, Map, Arrays** but some does such Linked lists, Queue when defining first, last elements. And also data source can be forced to be ordered with sort methods. Encountering order can slow down the effective of parallelism.

+ Call ‘**.unordered()’ to indicate encounter order is not meaningful to you.**

+ Ops like limit(), skip() and findFirst() will be optimized in the presence of unordered sources.

* **Merging Cost**

+ For some operations like doing sum, merging sub results is really cheap => no big deal.

+ For others like **groupingBy to a HashMap**, **it is insanely expensive as it involves lots of copying => Cost of merging overwhelms the parallelism advantage.**

Ex. Measuring IntStream.range(0, N).collect(toSet()). For n = 10k, approximately 4x slower when going parallel

**Summary:**

* **Before optimizing with parallelism, always …**

+ Have actual performance requirements.

+ Have reliable performance measurements