Stream Introduction.

Java 8 came with lot of new things. Few buzz words you might have heard can be Lambdas, closures, JavaScript Nashorn, Streams, and others. Yes, there are lot of exciting concepts introduced in Java 8. In this blog post I am going to discuss about Stream.

This is first article regarding Streams so please be patient.

Let us first take a use case and learn Streams from it.

Suppose I have a Person class with name and age as constructor parameters. Below is the code for Person class.

/\*\*

\* Currently I do not have any plans to extend any class so

\* making this class final.

\* \*/

**public** **final** **class** Person {

**private** **final** String name;

**private** **final** **int** age;

**public** Person(**final** String name, **final** **int** age) {

**this**.name = name;

**this**.age = age;

}

/\*\*

\* **@return** name

\* \*/

**public** String name() {

**return** name;

}

/\*\*

\* **@return** age

\* \*/

**public** **int** age() {

**return** age;

}

@Override

**public** String toString() {

StringBuilder sb = **new** StringBuilder();

sb.append("{")

.append(**this**.name()).append(":")

.append(**this**.age()).append("}");

**return** sb.toString();

}

}

Now you have List<Person> that will hold several Person objects. And now you want to get all Person with ***age greater than 35 sorted by name***.

So how to solve this scenario?

Well, it is simple isn’t it? One way to do this is:

Create a new List<Person> who will have age greater than 25.

List<Person> qualifiedPeople = **new** ArrayList<Person>();

**for** (**final** Person person : people) {

**if** (person.age() > 25) {

qualifiedPeople.add(person);

}

}

Now just sort the new List which we have named as qualifiedPeople

Collections.*sort*(qualifiedPeople, **new** Comparator<Person>() {

@Override

**public** **int** compare(**final** Person person1,

**final** Person person2) {

**return** person1.name().compareTo(person2.name());

}

});

And that’s it. You have *List<Person>* whose age is greater than 25 and sorted by name.

The problem with this approach is that we are saying or telling how to solve the problem. Couple of things to note here:

* Writing Imperative Code(Telling HOW? instead of WHY?). Read more [imperative code here](http://data-structure-learning.blogspot.com/2015/06/java-imperative-and-declarative.html).
* Doing iteration i.e. writing code that is not a use-case or writing Boiler-plate code.
* Writing anonymous inner class. Syntax can be confusing sometimes. And whenever you want to sort a List it will create a new Object.

There are only two lines from above that are important.

**if** (person.age() > 25) and person1.name().compareTo(person2.name())

Wouldn’t it be nice if the code can be reduced?

Let us write this code in Declarative way i.e. we just tell WHAT we want.

List<Person> p = *people*().stream()

.filter(person -> person.age() > 25)

.sorted(Comparator.*comparing*(person -> person.name()))

.collect(Collectors.*toList*());

In above lines of code much things are going on. Be patient.

* “->” is a lambda operator. Read What is [Functional Interface](http://data-structure-learning.blogspot.com/2015/06/functional-interfaces-java-8.html) and [Using Functional Interface](http://data-structure-learning.blogspot.com/2015/06/using-functional-interface.html) posts.
* Comparator has a method *comparing*. That’s weird right. Comparator is interface. Well *comparing* is default method in Comparator interface. We will talk about it later.
* Collector.toList() is a method that converts data from Stream to List.

If you see closely to the code, then you will know that there is no external iteration. No temporary variables. We are telling WHAT to do and not HOW to do.

This is a huge step in Java for Java Developers as we can now write code that will contain less temporary variables or objects.