**JAC444 - Lecture 8**

Functional Programming in Java

Segment 3 - Stream

**Objectives**

**Upon completion of this lecture, you should be able to:**

* Utilize Stream in Java
* Use Computed on Demand Principles
* Discover Principles of Functional Programming in Java

Stream

**In this lesson you will be learning about:**

* Definition of Stream
* Lambda Expression on Stream
* Functional Programming using Stream

# Stream Definition

## A *stream* is a sequence of elements (possibly-infinite) supporting sequential or parallel aggregate operations

## Characteriscs:

## 1. Each stream is used only once with:

1. Intermediate operations
2. Terminal operations

## 2. Operations act on entire stream (contrast with iterators)

## 3. There are two kinds of streams:

1. Sequential
2. Parallel

# Operations on Stream

There are two different kinds of operations

a. Intermediate operations

## Produce one stream from another

Example: **map, filter, stored, ...**

b. Terminal operations

## Extract a value or a collection from a stream

Example: **reduce, collect, count, findAny**

# Stream Sources

## One can generate Stream from different sources such as:

1. From any Collection: invoking stream() or parallelStream()

1. From BufferedReader: invoking lines()
2. From a function: Stream.generate(Supplier<T> s)

# Example of Stream

A stream obtained from a **collection** called **department**

## Example:

**// Calculate total salary of all fulltime employees using sum() final long fulltimeSalary = department**

**.stream() .filter(e -> (e.getEmpType() == Department.Type.FULLTIME)) .mapToInt(Employee::getEmpSalary)**

**.sum();**

# Design Decision

A stream facilitates 용이하게하다 programming using functional principles

## 1. Shorter - more expressiveness

## 2. More abstract - describes what to calculate (not how)

## 3. More efficient - avoids intermediate data structures

## 4. Runs in parallel - when requested with **parallelStream()**