# Lec 49 JAVA 8 P5:

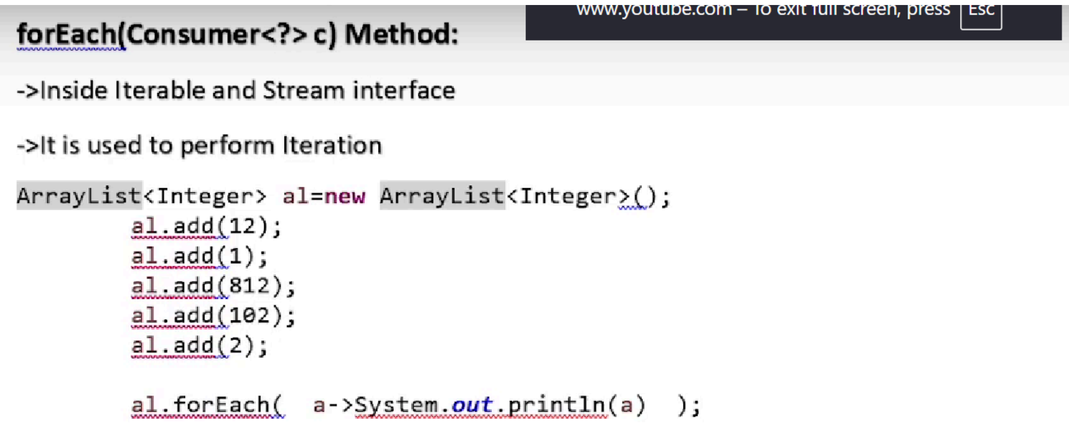
Java's Stream API, introduced in Java 8, allows you to process collections of data in a simpler and more efficient way using a functional, declarative style. It helps perform operations like filtering, mapping, sorting, and reducing data with less code and without changing the original data.

## Introduction to Stream API and forEach Method

Java 8 introduced the forEach method to make iterating over collections like ArrayList, LinkedList, and others simpler. The forEach method is available in both the Iterable and Stream interfaces and is commonly used with lambda expressions or functional interfaces such as Consumer.

## Usage and Benefits of forEach

The forEach method allows iteration over collections in a cleaner, more readable way compared to traditional for-loops or iterators. It takes a Consumer functional interface, which processes each element, reducing boilerplate code and improving clarity.



**forEach(Consumer<?> c) Method:**

->Inside Iterable and Stream interface

->It is used to perform Iteration

```
ArrayList<Integer> al=new ArrayList<Integer>();
        al.add(12);
        al.add(1);
        al.add(812);
        al.add(102);
        al.add(2);

        al.forEach(  a->System.out.println(a)  );
```

## Imperative vs Declarative Coding Styles

Imperative coding involves specifying every step of iteration and processing, leading to more lines of code. Declarative coding, as enabled by forEach and Stream API, focuses on what needs to be done rather than how, making code more concise and maintainable.

## Traditional Data Processing in Collections

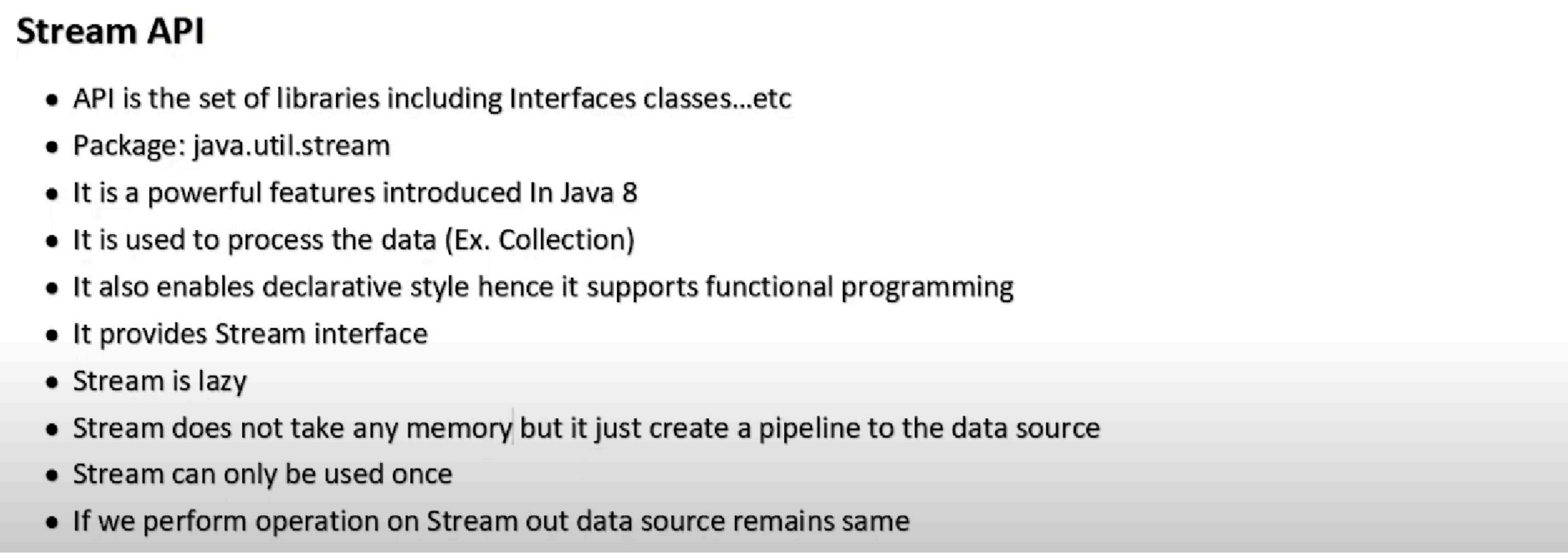
Before Stream API, processing collections (like doubling or filtering values) required manual loops, creating new lists for results, and extra logic for each operation, which made code verbose and error-prone.

## Real-World Collection Processing Example

In real-world scenarios, such as totaling product prices from a database, collections are used to hold data and multiple steps are needed to process, filter, and sum values. This can become complex and lengthy with traditional methods.

## Introduction to Stream API

The Stream API, added in Java 8, provides a set of interfaces and classes for processing data in a functional, declarative style. It is mainly used to process data from collections and arrays, allowing operations like filter, map, reduce, and count in a chainable way.

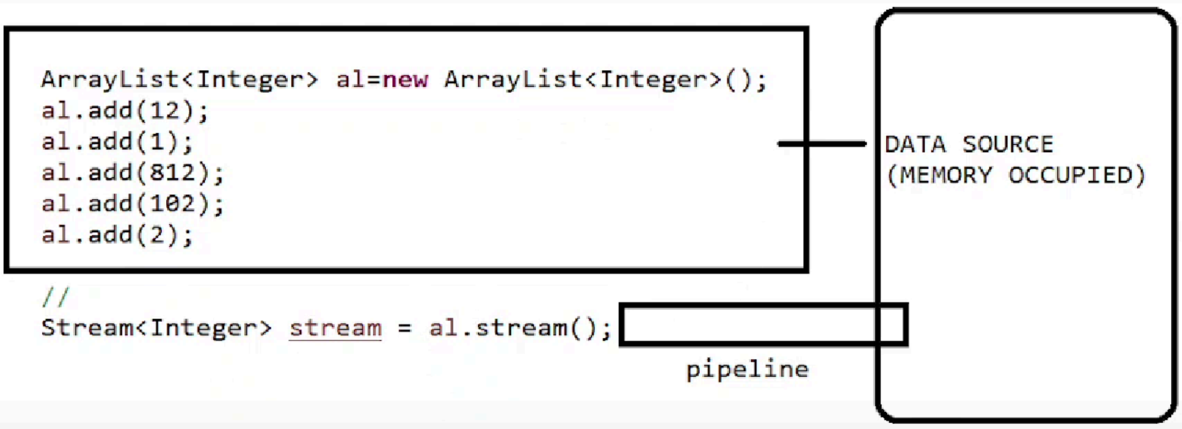


**Stream API**

- API is the set of libraries including Interfaces classes...etc
- Package: java.util.stream
- It is a powerful features introduced In Java 8
- It is used to process the data (Ex. Collection)
- It also enables declarative style hence it supports functional programming
- It provides Stream interface
- Stream is lazy
- Stream does not take any memory but it just create a pipeline to the data source
- Stream can only be used once
- If we perform operation on Stream out data source remains same

## How Streams Work Internally

When a stream is created from a collection, it does not copy or hold the data; instead, it creates a pipeline to the data source. The actual data is only processed when a terminal operation (like forEach or reduce) is called, a concept called "laziness."


```
ArrayList<Integer> al=new ArrayList<Integer>();
al.add(12);
al.add(1);
al.add(812);
al.add(102);
al.add(2);
//
Stream<Integer> stream = al.stream();
```

## Filtering and Mapping with Streams

The filter method selects elements based on a condition, while the map method transforms each element (e.g., doubling values). Each operation creates a new stream, and the original data remains unchanged.

## Stream Consumption and Single-Use Rule

Once a stream is consumed by a terminal operation, it cannot be reused. Attempting to use it again results in an exception. Each chained operation produces a new stream, and only the current stream can be operated on.

## Stream Operations Do Not Change Original Data

Performing operations on a stream does not alter the original collection; instead, each operation works on a new stream, ensuring the source data remains intact. This avoids the need for manual cloning.

## Chaining Multiple Stream Operations

Complex operations like multiplying, sorting, filtering, and summing elements can be performed in a single, readable statement using method chaining. Each intermediate operation returns a new stream, and the final result is obtained with a terminal operation.

## Using Reduce and Other Stream Methods

The reduce method combines elements of a stream into a single result, such as summing values. Other helpful methods include count, which returns the number of elements after processing. These methods make data processing concise and expressive.

## Key Takeaways and Practice Advice

The Stream API reduces code length, improves clarity, and enables functional programming techniques in Java. Understanding method return types and practicing lambda expressions are essential for mastering Stream API.