Sure! Let’s walk through your Java code line by line — explaining **each part**, from **package declaration** to **functional programming concepts**, **method references**, **lambda expressions**, and the **Streams API**.

**🔹 1. Package Declaration**

package Presentation;

* This declares the **package** name Presentation.
* Packages help organize Java classes into namespaces, just like folders.

**🔹 2. Imports**

import java.util.\*;

import java.util.function.\*;

import java.util.stream.Collectors;

* java.util.\* → Imports all utility classes like List, ArrayList, Random, etc.
* java.util.function.\* → Imports functional interfaces like Function, Predicate, Supplier, Consumer.
* java.util.stream.Collectors → Allows collecting stream output into Lists, Sets, etc.

**🔹 3. Person Class**

class Person {

private String name;

public Person(String name) {

this.name = name;

}

@Override

public String toString() {

return "Person{name='" + name + "'}";

}

}

* This is a simple class with one private field name.
* Constructor: Person(String name) sets the name.
* toString() is overridden to print a readable format for Person objects.

**🔹 4. Custom Functional Interface**

@FunctionalInterface

interface MathOperation {

int operate(int a, int b);

}

* A **Functional Interface** has only **one abstract method**.
* @FunctionalInterface ensures it can only have **one abstract method**.
* operate(int a, int b) defines a math operation.

**🔹 5. FunctionalDemo Class and Main Method**

public class FunctionalDemo {

public static void main(String[] args) {

* Main class that contains all the demo code.
* main(String[] args) is the program's entry point.

**🔸 Functional Interfaces in Action**

**✅ 1. Consumer<T> — takes input, returns nothing**

Consumer<String> printMessage = message -> System.out.println("Message: " + message);

printMessage.accept("Hello, Functional Interfaces!");

* Consumer<String> consumes a string.
* Lambda: message -> System.out.println(...)
* accept() is the method to call a Consumer.

**✅ 2. Supplier<T> — gives output, takes no input**

Supplier<Integer> randomNumberSupplier = () -> new Random().nextInt(100);

System.out.println("Random Number: " + randomNumberSupplier.get());

* Supplier<Integer> returns a random number.
* get() is used to fetch the value.

**✅ 3. Function<T, R> — input and return value**

Function<String, String> toUpperCase = str -> str.toUpperCase();

System.out.println("Uppercase: " + toUpperCase.apply("hello"));

* Function<String, String>: Takes a String, returns a String.
* apply() calls the function.

**✅ 4. Predicate<T> — boolean condition checker**

Predicate<Integer> isEven = num -> num % 2 == 0;

System.out.println("Is 10 even? " + isEven.test(10));

* Predicate<Integer>: Takes an Integer, returns boolean.
* test() evaluates the condition.

**✅ 5. Custom Functional Interface Usage**

MathOperation addition = (a, b) -> a + b;

System.out.println("Sum: " + addition.operate(5, 3));

* Implements the MathOperation interface using lambda.
* (a, b) -> a + b: adds two numbers.

**✅ 6. Combine Function + Consumer**

Function<Integer, Integer> squareFunction = x -> x \* x;

Consumer<Integer> printResult = result -> System.out.println("Square: " + result);

int number = 4;

printResult.accept(squareFunction.apply(number));

* squareFunction: returns square of a number.
* printResult: prints the result.

**✅ 7. Method Reference (Static Method)**

Function<Integer, Integer> methodRefSquare = FunctionalDemo::square;

System.out.println("Square using Method Reference: " + methodRefSquare.apply(5));

* FunctionalDemo::square refers to a static method.
* Equivalent to x -> square(x).

**✅ 8. Method Reference (Instance Method)**

FunctionalDemo demo = new FunctionalDemo();

Consumer<String> methodRefPrinter = demo::printInstanceMessage;

methodRefPrinter.accept("Hello from Method Reference!");

* Refers to a **non-static** method.
* demo::printInstanceMessage is a method reference.

**✅ 9. Method Reference (Arbitrary Object)**

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(System.out::println);

* System.out::println is a method reference to print each name.

**✅ 10. Constructor Reference**

Function<String, Person> constructorRef = Person::new;

Person p1 = constructorRef.apply("Alice");

System.out.println(p1);

* Person::new creates a new Person object.
* constructorRef.apply("Alice") → new Person("Alice").

**🔸 Streams API Usage**

List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50, 20, 10, 60, 70, 80, 90, 100);

* Create a list of integers.

**➤ filter() → Only elements > 30**

System.out.println("Filtered (n > 30): " + numbers.stream().filter(n -> n > 30).collect(Collectors.toList()));

**➤ map() → Multiply each by 2**

System.out.println("Mapped (n \* 2): " + numbers.stream().map(n -> n \* 2).collect(Collectors.toList()));

**➤ sorted() → Sort ascending**

System.out.println("Sorted: " + numbers.stream().sorted().collect(Collectors.toList()));

**➤ distinct() → Remove duplicates**

System.out.println("Distinct: " + numbers.stream().distinct().collect(Collectors.toList()));

**➤ limit(n) → Take first n elements**

System.out.println("Limited (First 5 numbers): " + numbers.stream().limit(5).collect(Collectors.toList()));

**➤ skip(n) → Skip first n elements**

System.out.println("Skipped (After first 3 numbers): " + numbers.stream().skip(3).collect(Collectors.toList()));

**➤ count() → Count elements > 50**

System.out.println("Count of numbers > 50: " + numbers.stream().filter(n -> n > 50).count());

**➤ forEach() → Print each number**

numbers.stream().forEach(n -> System.out.print(n + " "));

**➤ collect(Collectors.toSet()) → Collect to a Set**

System.out.println("Collected Set: " + numbers.stream().collect(Collectors.toSet()));

**➤ reduce() → Sum of all numbers**

System.out.println("Sum of all numbers: " + numbers.stream().reduce(0, (a, b) -> a + b));

**➤ anyMatch() → Check if any number > 80**

System.out.println("Is there a number > 80? " + numbers.stream().anyMatch(n -> n > 80));

**🔸 Static & Instance Method Definitions**

public static int square(int number) {

return number \* number;

}

public void printInstanceMessage(String message) {

System.out.println(message);

}

* square is a static method to return square.
* printInstanceMessage is a non-static method to print.

**✅ Summary of Key Concepts**

| **Feature** | **Used For** | **Method** |
| --- | --- | --- |
| Consumer<T> | Takes input, no return | accept() |
| Supplier<T> | No input, gives value | get() |
| Function<T, R> | Input & output | apply() |
| Predicate<T> | Tests a condition | test() |
| Custom Interface | Custom logic | operate() |
| Method Reference | Replaces lambda | ::methodName |
| Streams | Data processing | filter, map, reduce, etc. |

Would you like a diagram or flowchart for visual understanding?

Absolutely! Let's go through the entire code step-by-step, explaining **each class, method, and stream operation** in detail — especially focusing on **Java Stream API** and how it's used functionally in your program.

**📦 Package: Presentation**

All the classes are declared inside the Presentation package. This is a **namespace mechanism in Java** used to group related classes. The folder name should match this package name when compiling/running.

**🧱 Class 1: Course**

public class Course {

private String courseName;

private String courseId;

private int credit;

**🔹 Fields:**

* courseName – name of the course (e.g., "JAVA")
* courseId – unique ID like "CSE4001"
* credit – number of credits the course carries (like 3 or 2)

**🔹 Constructor:**

public Course(String courseName, String courseId, int credit)

Initializes a course with values passed when the object is created.

**🔹 Getters and Setters:**

Standard Java methods to access and modify private variables.

**🔹 toString():**

Returns a formatted string representation of the Course object.

**🧱 Class 2: Student**

public class Student {

private int id;

private String name;

private List<Course> courses;

private double cgpa;

**🔹 Fields:**

* id: student ID (e.g., 1, 2)
* name: student name
* courses: list of Course objects the student is enrolled in
* cgpa: cumulative GPA

**🔹 Constructor:**

public Student(int id, String name, List<Course> courses, double cgpa)

Initializes a student with given attributes.

**🔹 Getters and Setters:**

Used to access/modify each attribute.

**🔹 toString():**

Returns string like Student [id=1, name=Akash, cgpa=8.3].

**🧠 Class 3: Stream**

This is the class containing your **main method** and all the **Java Stream API operations**.

**🔥 The main() method:**

public static void main(String[] args) {

**🧑‍🎓 Students List Initialization**

List<Student> students = Arrays.asList(

new Student(1, "Akash", Arrays.asList(new Course("JAVA", "CSE4001",3), new Course("C++", "CSE4006",2)), 8.3),

...

);

You're creating a **hardcoded list** of 6 students, each with their enrolled courses and CGPA.

**💡 Now comes the magic — Java Streams**

A **Stream** is a sequence of elements supporting sequential and parallel aggregate operations. Think of it like a pipeline where you can:

* Filter
* Map
* Reduce
* Collect
* Iterate

Streams **don’t store data**. They operate on a data source (like a list) and perform computation lazily.

**1. Print all student names**

students.stream()

.map(Student::getName)

.forEach(System.out::println);

* .stream() → creates a stream from the student list.
* .map(Student::getName) → transforms each Student into their name.
* .forEach(...) → prints each name.

**2. Filter students with CGPA > 8**

students.stream()

.filter(s -> s.getCgpa() > 8)

.forEach(System.out::println);

* .filter(...) → only keep students whose CGPA is > 8
* s -> s.getCgpa() > 8 is a **lambda function**
* Prints the matching Student objects.

**3. Count total number of students**

long count = students.stream().count();

Simple count of elements in the stream.

**4. Find the student with the highest CGPA**

Student topper = students.stream()

.max(Comparator.comparing(Student::getCgpa))

.orElse(null);

* .max(...) → returns the student with max CGPA using comparator.
* orElse(null) → returns null if stream is empty.

**5. Calculate average CGPA**

double avgCgpa = students.stream()

.collect(Collectors.averagingDouble(Student::getCgpa));

* Uses Collectors.averagingDouble() to compute average CGPA.

**6. Group students by CGPA**

Map<Double, List<Student>> groupedByCgpa = students.stream()

.collect(Collectors.groupingBy(Student::getCgpa));

* Groups students who have the same CGPA.
* Returns a Map<Double, List<Student>>.

**7. All course names (no duplicates)**

students.stream()

.flatMap(s -> s.getCourses().stream())

.map(Course::getCourseName)

.distinct()

.forEach(System.out::println);

* .flatMap(...) → flattens nested lists (like student → multiple courses)
* .map(Course::getCourseName) → gets the course name
* .distinct() → removes duplicates
* Prints all unique course names.

**8. Map of student name → CGPA**

Map<String, Double> nameToCgpa = students.stream()

.collect(Collectors.toMap(Student::getName, Student::getCgpa));

* Converts to a map: name as key, CGPA as value.
* If duplicate names exist, this could throw error (see note below).

🔴 **Note**: Student names like "Prajwal" and "John" with same ID (4) could cause duplicate key issue here.

**9. Total credits per student**

students.stream()

.collect(Collectors.toMap(

Student::getName,

s -> s.getCourses().stream().mapToInt(Course::getCredit).sum()

))

.forEach((name, credits) -> System.out.println(name + ": " + credits + " credits"));

* Maps each student name to the **sum of their course credits**.
* Uses mapToInt(...).sum() to compute total credits per student.

**10. Join all student names into a string**

String joinedNames = students.stream()

.map(Student::getName)

.collect(Collectors.joining(", "));

* Converts names into a single string, separated by commas.

**11. List of all course IDs (distinct)**

List<String> courseIds = students.stream()

.flatMap(s -> s.getCourses().stream())

.map(Course::getCourseId)

.distinct()

.collect(Collectors.toList());

* Similar to course names, but this time gets unique **course IDs**.

**⚠️ Optional Improvements**

* Handle duplicate student names in .toMap(...) using:

.collect(Collectors.toMap(

Student::getName,

Student::getCgpa,

(existing, replacement) -> replacement)) // resolve conflict

**🏁 Summary of Java Stream Methods Used:**

| **Method** | **What it does** |
| --- | --- |
| .stream() | Turns a collection into a stream |
| .map() | Transforms each element |
| .filter() | Filters elements by a condition |
| .forEach() | Performs an action on each element |
| .count() | Counts elements |
| .max() | Finds max element by comparator |
| .collect() | Collects results (e.g., to list/map) |
| .groupingBy() | Groups elements by key |
| .flatMap() | Flattens nested collections |
| .distinct() | Removes duplicates |
| .joining() | Joins strings |

If you want, I can convert this code into a **menu-driven console app**, or even add a FunctionalDemo class for better separation. Want me to do that next?