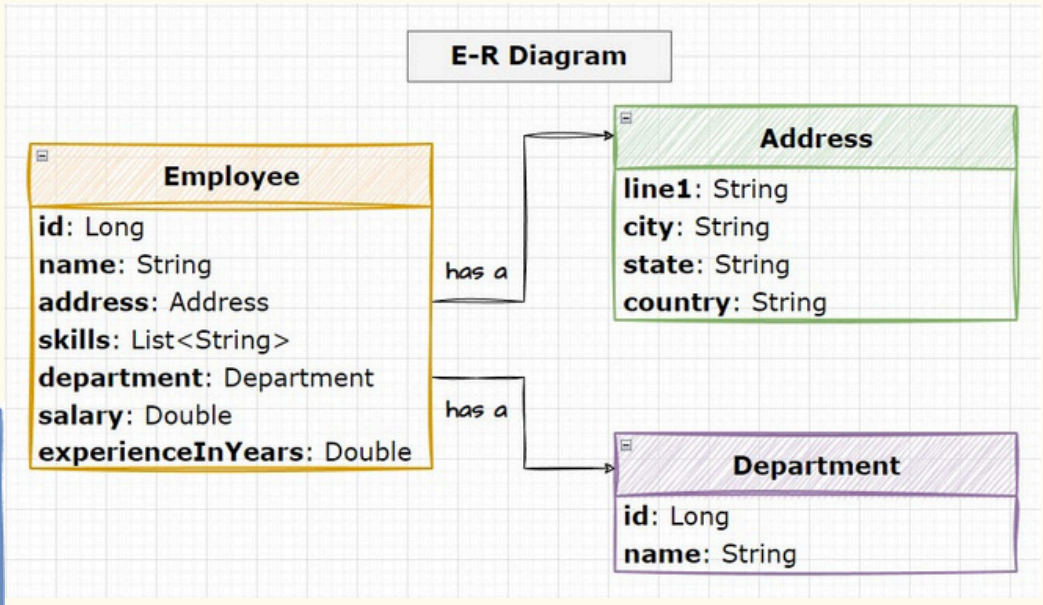
STREAM OPERATIONS – 2

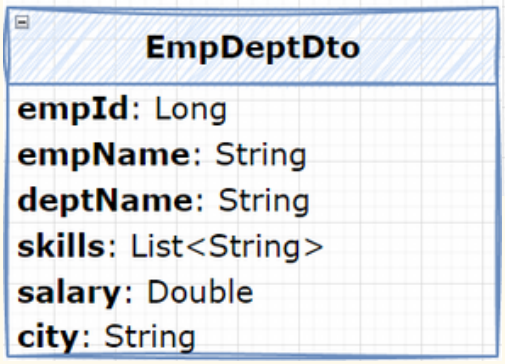
Java 8+

Quick Recap – *Project Structure:*

Model Classes:

1. Employee
2. Department
3. Address





EmpDeptDto: DTO class having some fields.

Repository Layer:

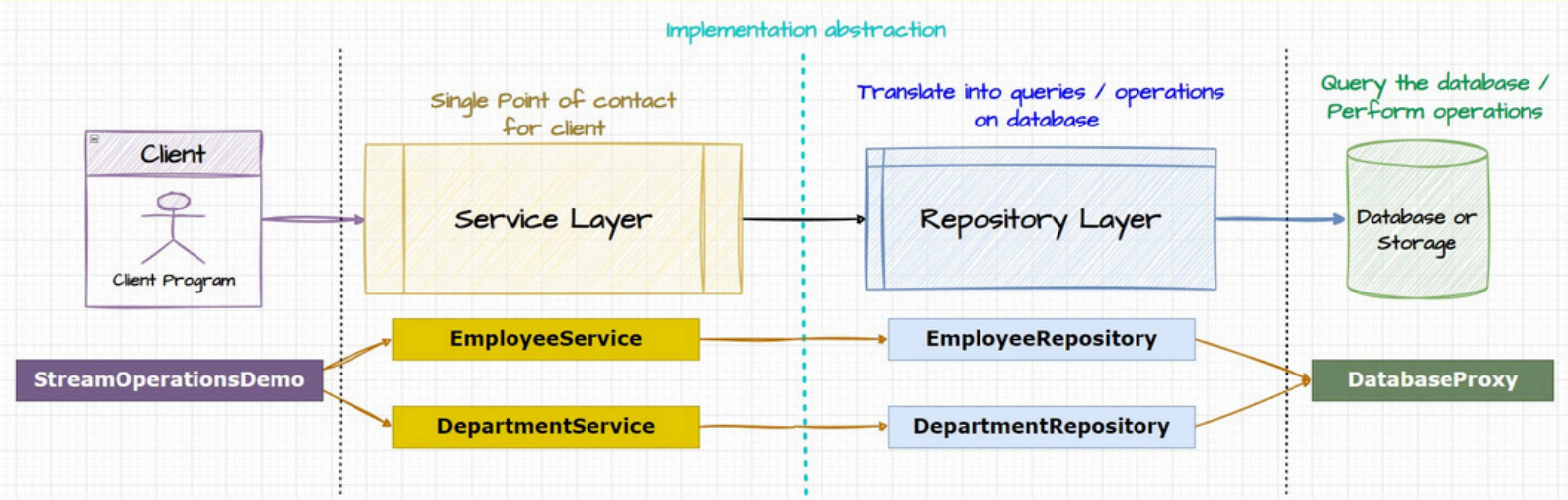
1. EmployeeRepository
2. DepartmentRepository

Service Layer:

1. EmployeeService
2. DepartmentService

Database: DatabaseProxy class consists of static data

Project Structure:



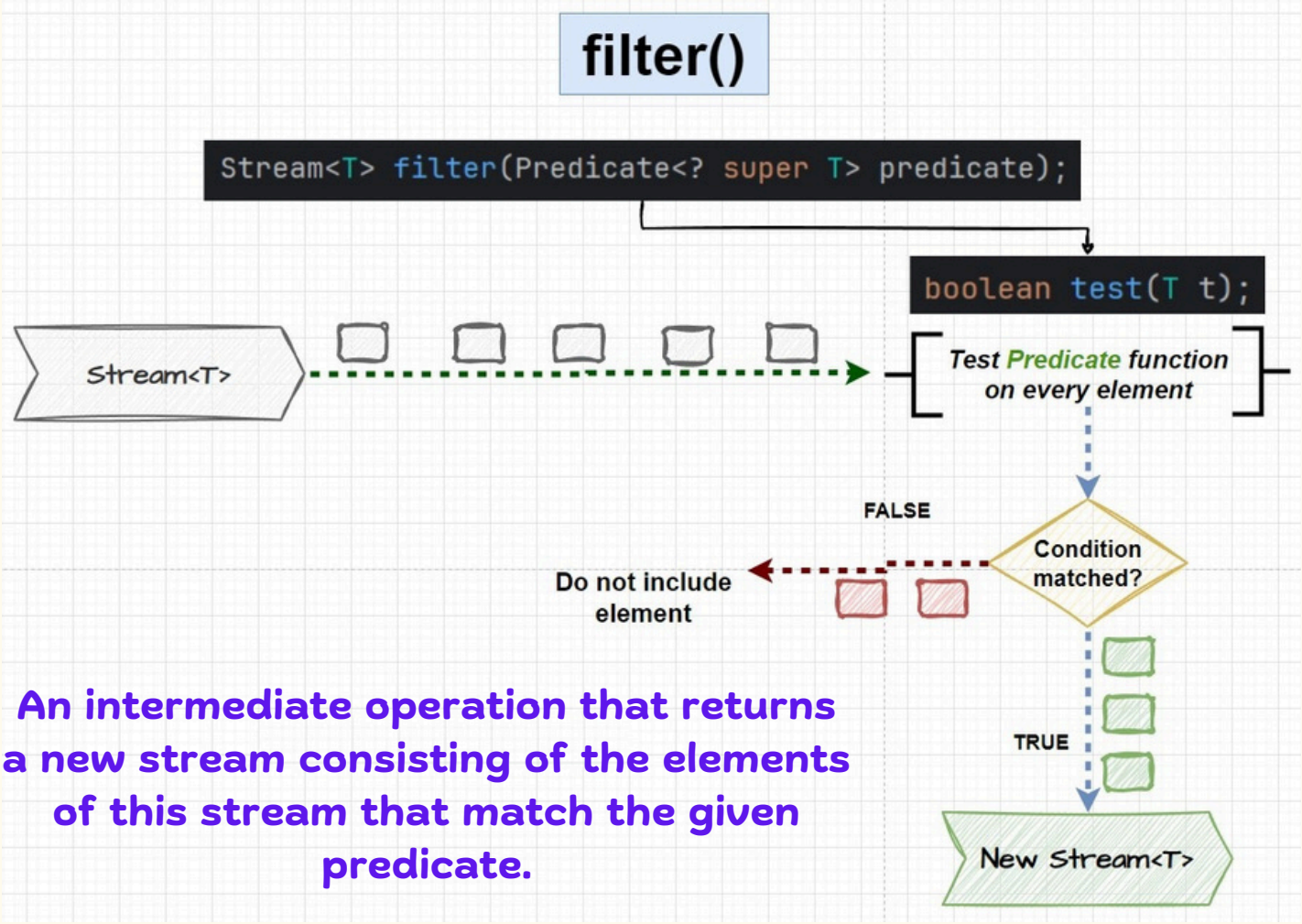
#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

Quick recap –

Stream Intermediate Operation – filter



Example: *Get list of employees in given department having salary greater than given salary – Repository Layer*

```
DatabaseProxy.getEmployees() List<Employee>
 .stream() Stream<Employee>
 .filter(e -> e.getDepartment().getId() == deptId
                 && e.getSalary() > minSalary)
 .collect(Collectors.toList());
```

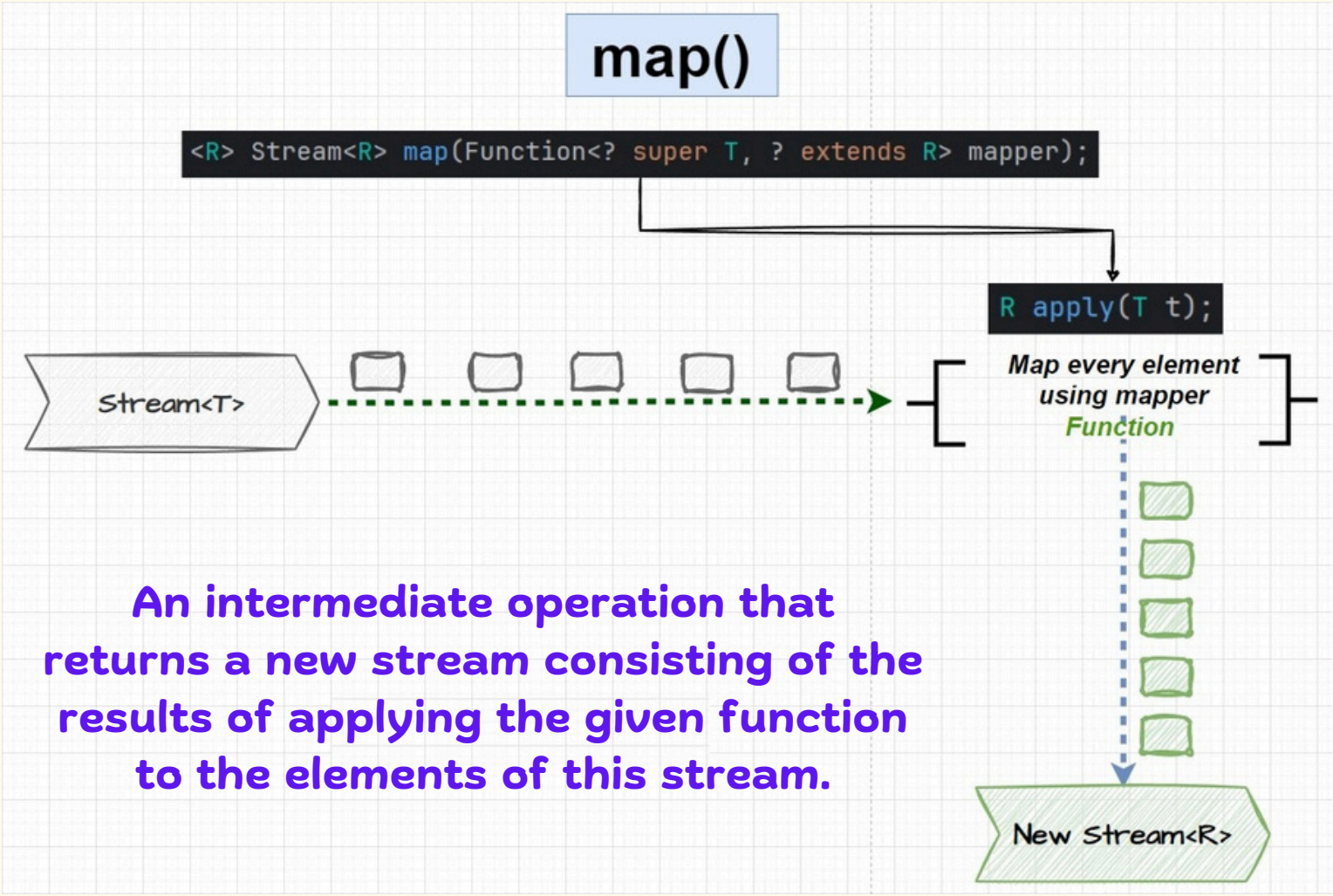
#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

Quick recap –

Stream Intermediate Operation – map



Example: *Get list of employee DTOs in given department having salary greater than given salary - Service Layer*

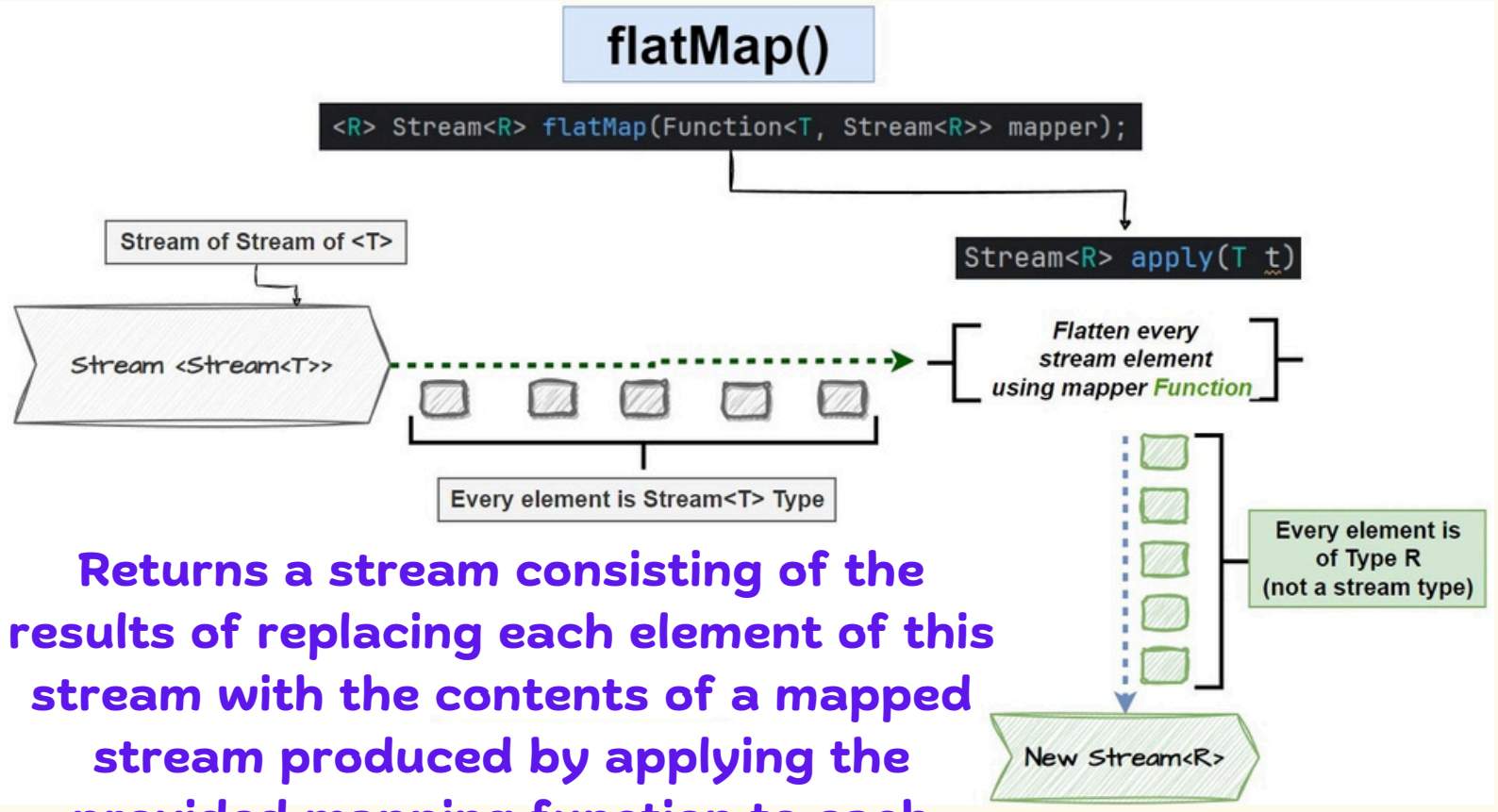
```
employeeRepository
  .findByDepartmentIdAndSalaryGreaterThan(deptId, minSalary)
  .stream()  Stream<Employee>
  .map(EmpDeptDto::new)  Stream<EmpDeptDto>
  .collect(Collectors.toList());
```

STREAM OPERATIONS – 2

Java 8+

Quick recap –

Stream Intermediate Operation – flatMap



Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

Example: Get all skills employees in a given department have - Service Layer

```
employeeRepository.findSkillsByDeptId(deptId)
  .stream() Stream<Employee>
  .flatMap(e -> e.getSkills().stream()) Stream<
  .distinct()
  .collect(Collectors.toList());
```

#CodingWalaShree

STREAM OPERATIONS – 2

🚀 Java Streams Sorting Like a Pro! 🔥 sorted() with Comparable & Comparator | PART 4

Java 8+

Sorting Streams

Sorting is a fundamental operation in programming, often required when dealing with collections of data. In Java, the Stream API provides a convenient way to perform sorting using the **sorted()** method.

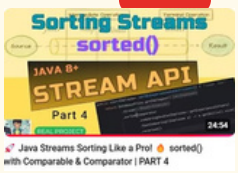
The sorted() method

1. **sorted() (Natural Order)**: Uses the natural ordering of elements, requiring them to implement the Comparable interface.
2. **sorted(Comparator c) (Custom Order)**: Accepts a Comparator to define a custom sorting order.

Primitive Wrapper Types, Strings, and Other Predefined Comparable Types

Primitive wrapper classes in Java (e.g., Integer, Double), String, and other predefined types (e.g., LocalDate, BigDecimal) already implement the Comparable interface. Therefore, sorting a list of these types works seamlessly:

STREAM OPERATIONS – 2



Java 8+

Example: *Applying sorted() on Integers, Strings and BigDecimals*

```
System.out.println("Applying sorted() on Integers..");
List<Integer> integers = Stream.of( ...values: 2, 5, 1, 3)
        .sorted()
        .collect(Collectors.toList());
System.out.println("Sorted Integers: " + integers);

System.out.println("Applying sorted() on Strings..");
List<String> strings = Stream.of( ...values: "Orange", "Apple", "Pineapple", "Kiwi")
        .sorted()
        .collect(Collectors.toList());
System.out.println("Sorted Strings: " + strings);

System.out.println("Applying sorted() on BigDecimals..");
List<BigDecimal> bigDecimals = Stream.of(BigDecimal.valueOf(10.5),
                BigDecimal.valueOf(7.5),
                BigDecimal.valueOf(30.5))
        .sorted()
        .collect(Collectors.toList());
System.out.println("Sorted BigDecimals: " + bigDecimals);
```

Output:

```
Applying sorted() on Integers..
Sorted Integers: [1, 2, 3, 5]
Applying sorted() on Strings..
Sorted Strings: [Apple, Kiwi, Orange, Pineapple]
Applying sorted() on BigDecimals..
Sorted BigDecimals: [7.5, 10.5, 30.5]
```

#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

Comparator vs Comparable

Attempting to Sort Without Implementing Comparable

If we attempt to sort a custom class that does not implement Comparable, a **ClassCastException** is thrown.

Example: *Applying sorted() on Employees class (not implementing Comparable)*

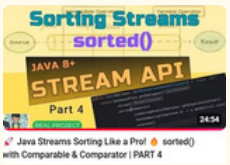
```
public class Employee {  45 usages   codingwalashree *
    /**
     * Member variables of Employee class
     * */
    private long id;  4 usages
    private String name;  3 usages
    private Address address;  4 usages

    private List<String> skills;  4 usages

    private Department department;  4 usages
    private double salary;  4 usages
    private double experienceInYears = 0;  4 usages
```

```
System.out.println("Applying sorted() on Employees (not implementing Comparable)..");
DatabaseProxy.getEmployees() List<Employee>
        .stream() Stream<Employee>
        .sorted()
        .forEach(System.out::println);
```

Output:

```
Applying sorted() on Employees (not implementing Comparable)..
Exception in thread "main" java.lang.ClassCastException Create breakpoint : org.cws
    at java.util.Comparators$NaturalOrderComparator.compare(Comparators.java:4
    at java.util.TimSort.countRunAndMakeAscending(TimSort.java:355)
```

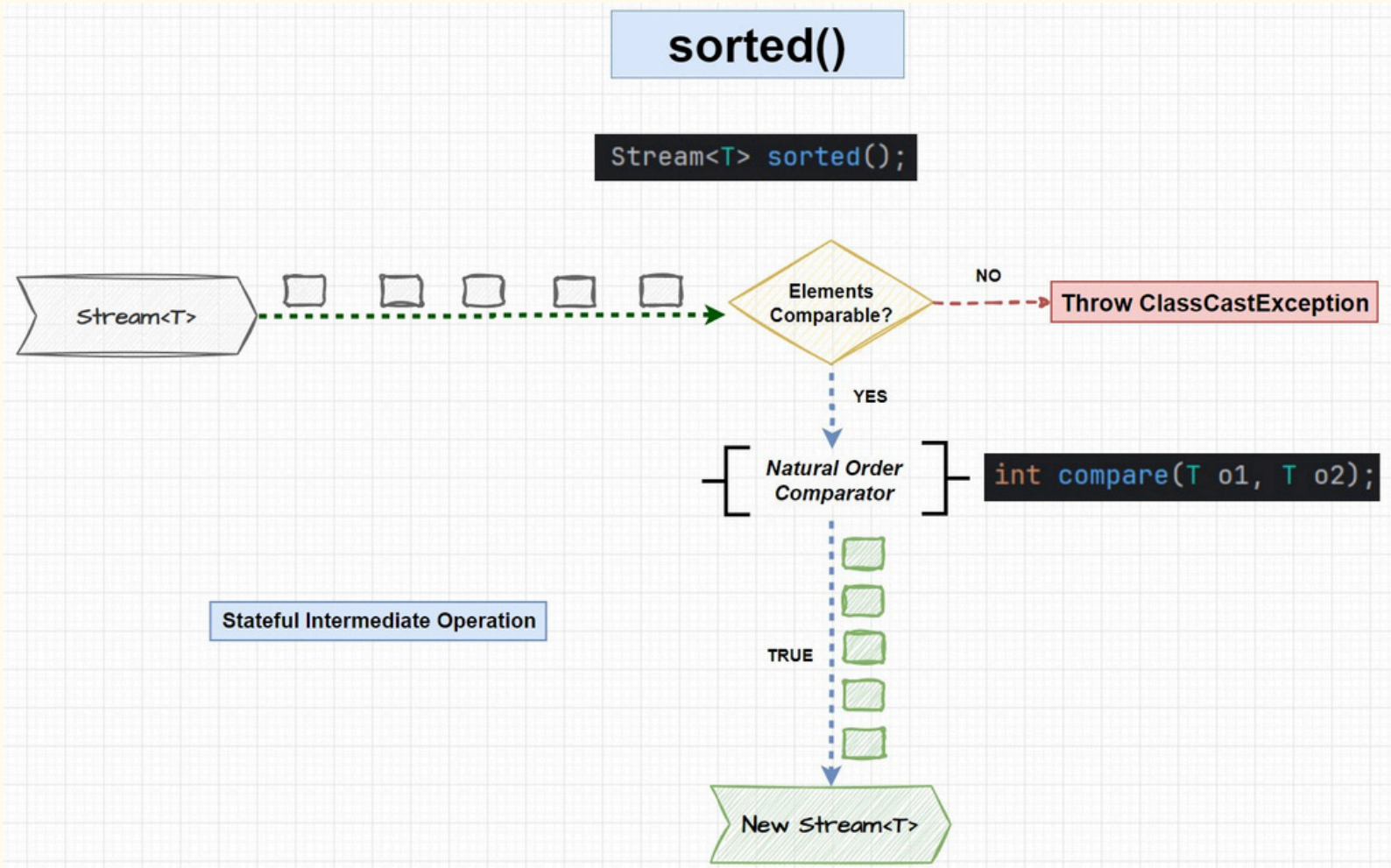
STREAM OPERATIONS – 2



Java 8+

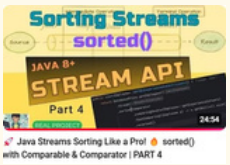
Attempting to Sort after Implementing Comparable

- Implement **Comparable** interface and override **compareTo()** method to **enable natural ordering** .
- Operations like **sorting** and **grouping** are based on **ordering criteria**. When you implement Comparable interface, you provide your class capability to order instances of it's own type based on **ordering key**.
- sorted() method without any argument can be used for natural sorting



#CodingWalaShree

STREAM OPERATIONS – 2



Java 8+

Example: *Applying sorted() on Employees class after implementing Comparable – sorting on employee ID*

```
public class Employee implements Comparable<Employee> {
    @Override  new *
    public int compareTo(Employee other) {
        return Long.compare(this.getId(), other.getId());
    }
    /**
     * Member variables of Employee class
     * */
```

```
System.out.println("Applying sorted() on Employees (after implementing Comparable)..");
DatabaseProxy.getEmployees() List<Employee>
        .stream() Stream<Employee>
        .sorted()
        .forEach(System.out::println);
```

Output:

```
Applying sorted() on Employees (after implementing Comparable)..
Employee{id=1, name='Rajesh', address=Address{line1='address1', city='Mumbai', state='Maharasht
Employee{id=2, name='Shiven', address=Address{line1='address2', city='Pune', state='Maharashtra
Employee{id=3, name='Preeti', address=Address{line1='address3', city='Mumbai', state='Maharasht
Employee{id=4, name='Avani', address=Address{line1='address4', city='Mumbai', state='Maharashtr
Employee{id=5, name='Santosh', address=Address{line1='address5', city='Pune', state='Maharashtr
Employee{id=6, name='Snehal', address=Address{line1='address6', city='Pune', state='Maharashtra
```

#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

Comparator Interface

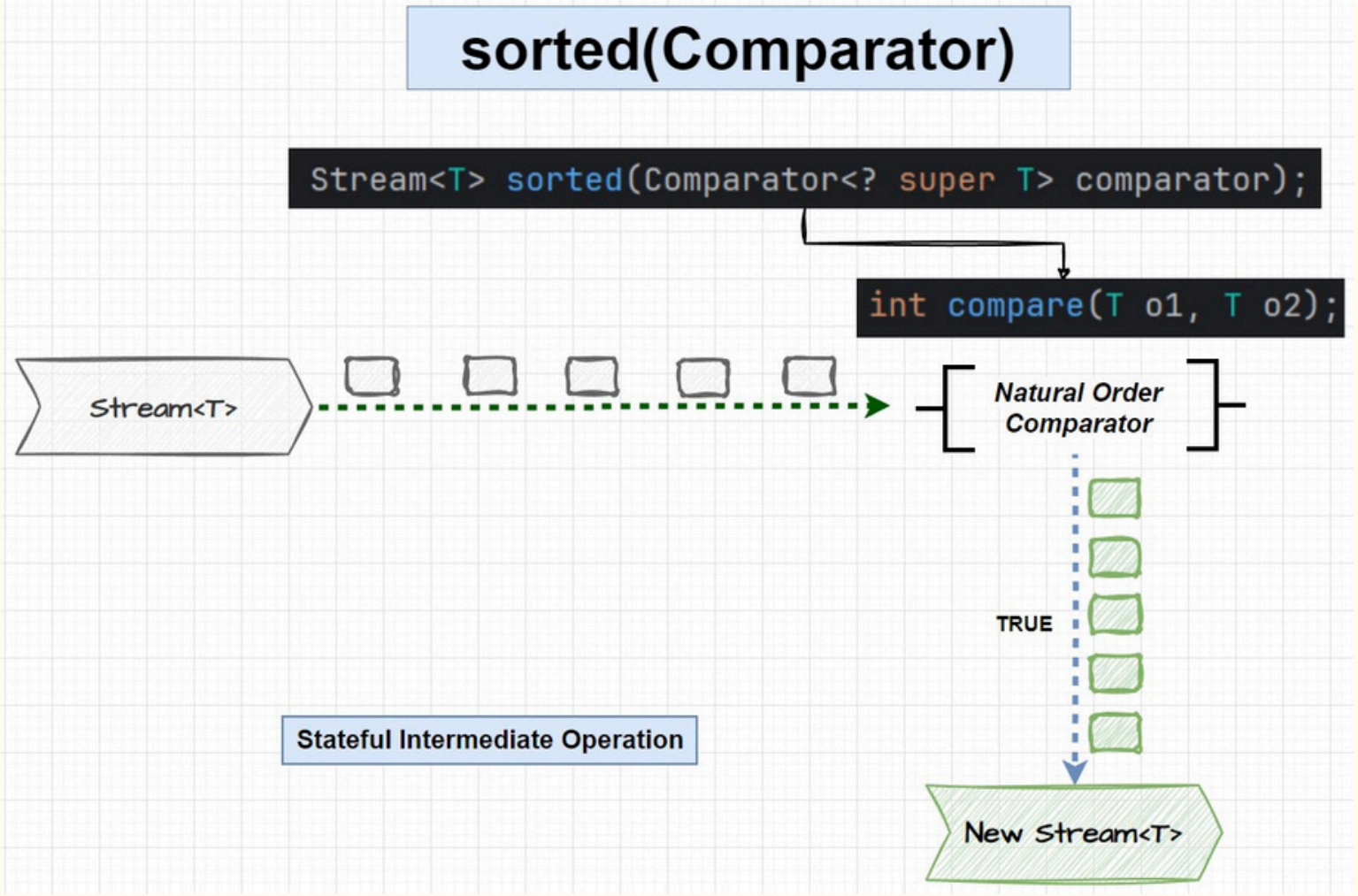
- Used for defining **multiple sorting criteria**.
- Requires implementing **compare()** method.
- Does **not** require modifying the original class.
- Operations like **sorting** and **grouping** are based on **ordering criteria**. By implementing the **Comparator** interface, you can define **multiple sorting criteria externally**, allowing **flexible ordering** of elements for a given class.
- **Comparator** allows **chaining** comparators using factory methods like **thenComparing()**, enabling **multi-level sorting** where elements are first compared by one attribute and, if equal, compared by another.

#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

The sorted() with Comparator



Example: Sort all employees by experience in years in ascending fashion.

```
DatabaseProxy.getEmployees() List<Employee>
  .stream() Stream<Employee>
  .sorted(Comparator.comparing(Employee::getExperienceInYears))
  .collect(Collectors.toList());
```

Output:

```
 sort by experience ascending -----
{id=15, name='Jitesh', skills=[Accounting], experienceInYears=1.5}
{id=3, name='Preeti', skills=[React, Angular, NodeJS], experienceInYears=2.0}
{id=1, name='Rajesh', skills=[Java, Spring Boot, Hibernate], experienceInYears=3.0}
{id=2, name='Shiven', skills=[Java, Spring Boot, Hibernate, JavaScript, HTML, CSS, HTTP, React], experienceInYears=3.0}
{id=11, name='Riya', skills=[Java, Spring Boot, Hibernate], experienceInYears=4.0}
```

#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

Example: Sort all employees by experience in years in descending fashion.

```
DatabaseProxy.getEmployees() List<Employee>
  .stream() Stream<Employee>
  .sorted(
          Comparator.comparingDouble(Employee::getExperienceInYears)
                  .reversed()
  )
  .collect(Collectors.toList());
```

Output:

```
sort by experience descending -----
{id=9, name='Nishant', skills=[Recruitment, Screening, Onboarding, People Management], experienceInYears=15.0}
{id=8, name='Sakshi', skills=[Recruitment, Screening, Onboarding], experienceInYears=13.5}
{id=12, name='Atharva', skills=[Java, Spring Boot, Hibernate, JavaScript, HTML, CSS, HTTP, Angular], experienceInYears=12.0}
{id=10, name='Vivansh', skills=[Java, Spring Boot, Hibernate, JavaScript, HTML, CSS, HTTP, React], experienceInYears=11.0}
{id=7, name='Pradeep', skills=[Java, Spring Boot, Hibernate, JavaScript, HTML, CSS, HTTP, Angular], experienceInYears=10.0}
{id=13, name='Preeti', skills=[Accounting, Taxation and Compliance, Risk Management], experienceInYears=10.0}
```

- Comparator's comparing(), comparingDoouble(), **thenComparing()**, etc. methods take a lambda to extract a sorting key.
- In this example, experience in years is a ordering key on which we want to sort the employes. So, in **comparingDouble()** , employee instance's **getExperienceInYears()** method acts as a key extractor function.
- **reversed()** method reverses the order of elements sorted by applying first Comparator i.e. sorts in descending fashion.

#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

Example: Sort all employees by skill count and experience in ascending fashion.

```
DatabaseProxy.getEmployees() List<Employee>
  .stream() Stream<Employee>
  .sorted(Comparator
          .comparingDouble(Employee::getExperienceInYears)
          .thenComparing((Employee e) -> e.getSkills().size())
          .reversed()
  )
  .collect(Collectors.toList());
```

Output:

```
name='Santosh', skills=[Recruitment, Screening, Onboarding], experienceInYears=5.0}
name='Avani', skills=[Java, Spring Boot, Hibernate, JavaScript, HTML, CSS, HTTP, Angular], experienceInYears=4.5}
, name='Riya', skills=[Java, Spring Boot, Hibernate], experienceInYears=4.0}
name='Shiven', skills=[Java, Spring Boot, Hibernate, JavaScript, HTML, CSS, HTTP, React], experienceInYears=3.0}
name='Rajesh', skills=[Java, Spring Boot, Hibernate], experienceInYears=3.0}
```

- In this example, experience in years is a first ordering key on which we want to sort the employes. So, in **comparingDouble()** , employee instance's **getExperienceInYears()** method acts as a key extractor function.
- Second ordering key is number of skills of the employee.
- **reversed()** method reverses the order of elements sorted by applying first and second Comparator.

#CodingWalaShree

STREAM OPERATIONS – 2

Java 8+

Performance Impact of sorted()

Sorting in Java Streams is stateful intermediate operation and is performed lazily. The performance impact depends on several factors:

- **Time Complexity:** Has a worst-case complexity of O(n log n), but it's adaptive and can be O(n) for nearly sorted data.
- **Memory Usage:** Sorting operations require additional memory for merging and reordering.
- **Parallel Streams:** Sorting in parallel streams may not always be efficient due to partitioning and merging overhead.

For **large datasets**, consider:

- Performing sorting at the **database level** (SQL order by) rather than in-memory to improve efficiency.
- Using **parallel streams cautiously**, as sorting requires partitioning + merging sorted partitions.
- Avoiding **unnecessary sorting** – if already sorted upstream, don’t re-sort.

STREAM OPERATIONS – 2

Java 8+

Examples in this presentation are covered in my YouTube Video on Stream API Part 4🚀

🚀 Java Streams Sorting Like a Pro! 🔥 sorted() with Comparable & Comparator | PART 4

Note: *As I cover more Stream operations in upcoming videos on CodingWalaShree, I'll update this presentation and share it on LinkedIn – so stay tuned! 🚀*

To be continued. . .

#CodingWalaShree