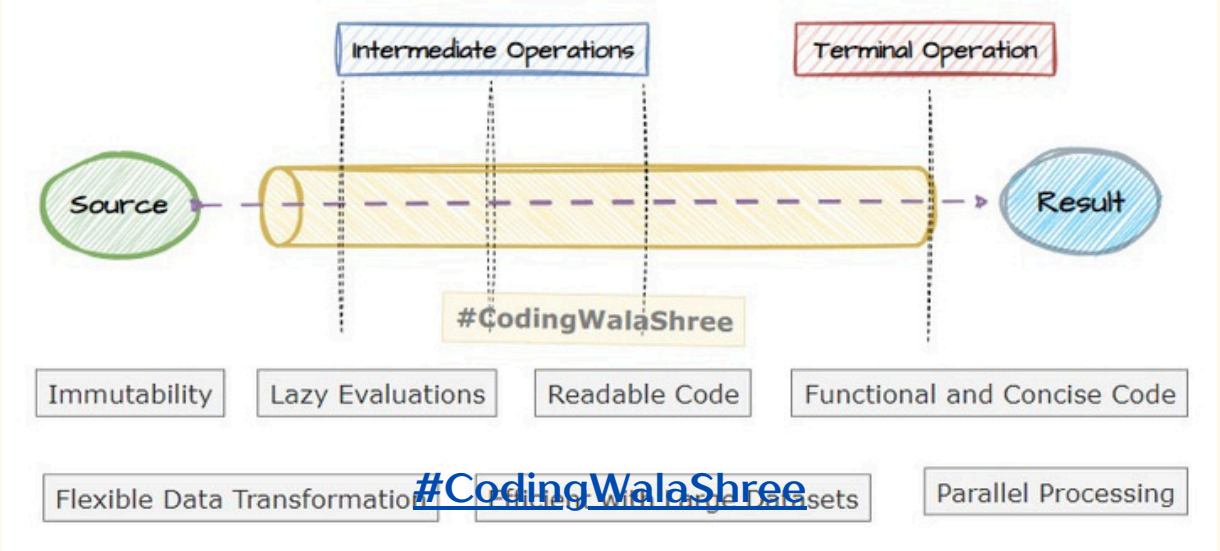
STREAM OPERATIONS – I

Master Stream API - Game Changing Java 8 Feature - PART 1

Java 8+

Quick recap – What are Streams?

- A stream is a sequence of data items that are conceptually produced one at a time.
- Source of stream: Collections (List / Set), Arrays, a Generator Function or I/O resources like Files can be at the source of a stream.
- A stream is a conceptually fixed data structure (you can't add or remove elements from it) whose elements are computed on demand.
- A stream cannot modify the underlying source.
- Streams make use of internal iterations: the iteration is abstracted away through operations such as filter, map, and sorted.
- The code is written in a declarative way: you specify what you want to achieve (that is, filter dishes that are low in calories) as opposed to specifying how to implement an operation (using control-flow blocks such as loops and if conditions).



STREAM OPERATIONS – I

Java 8+

Quick recap - 9 ways to create Streams

1. Empty stream
2. From array
3. From Collection - List & Set
4. Using Stream Builder -- Stream.builder()
5. Using static method of() -- Stream.of()
6. From generator function -- Stream.generate()
7. From static method iterate -- Stream.iterate()
8. Stream of primitive types -- IntStream, LongStream and DoubleStream
9. By reading a file -- Files.lines()

Watch videos on my channel @CodingWalaShree on creating streams in 9 ways and filter-map-flatMap with examples:

9 Ways to Create Streams in Java 8 - PART 2

🚀 Master Java Streams: filter(), map(), and flatMap() with Real-Project Examples!

#CodingWalaShree

STREAM OPERATIONS – I

Java 8+

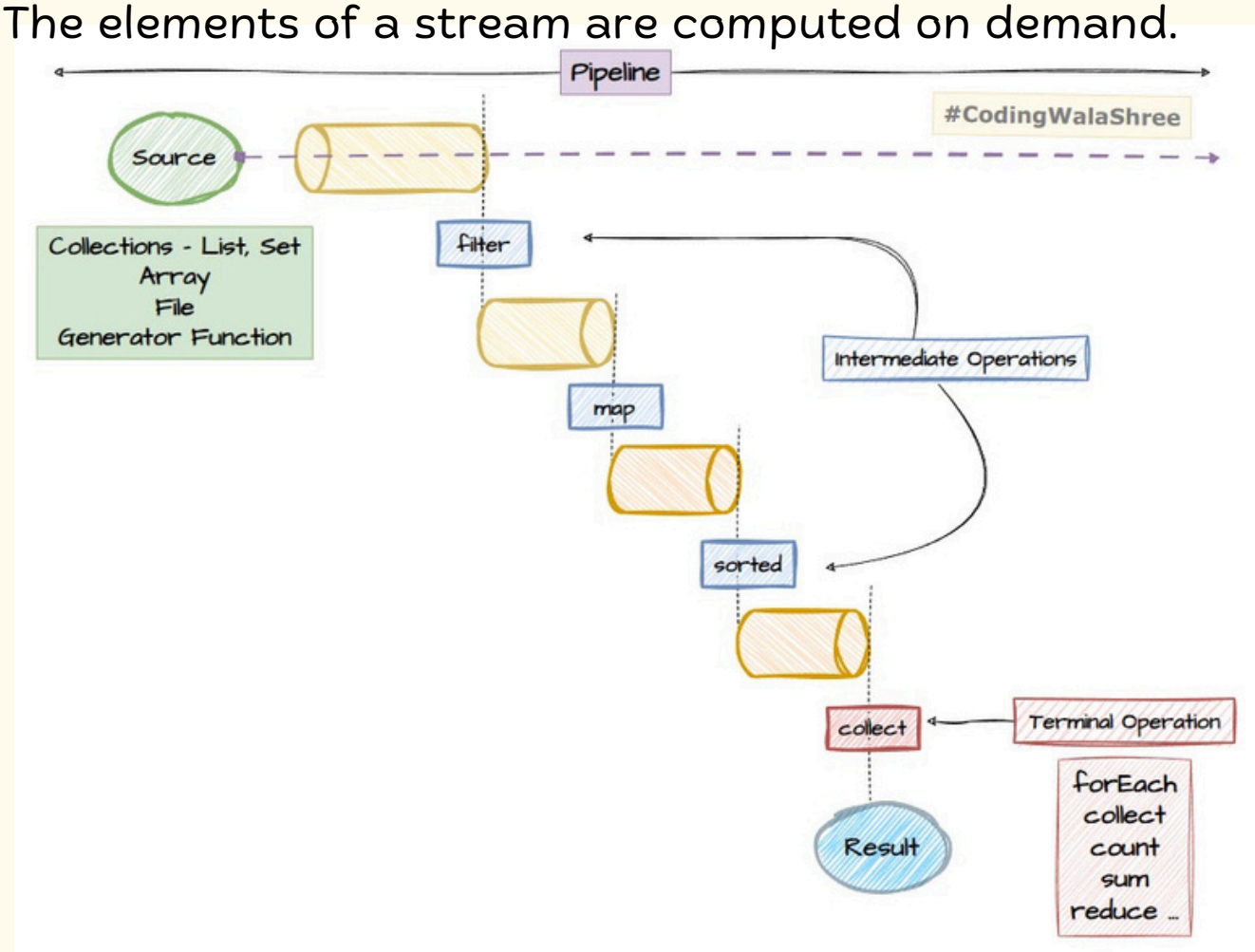
Types of Stream Operations

- There are two types of stream operations: intermediate and terminal operations.

- **Intermediate operations** such as filter and map return a stream and can be chained together. They're used to set up a pipeline of operations but don't produce any result.

- **Terminal operations** such as forEach and count return a non-stream value and process a stream pipeline to return a result.

- The elements of a stream are computed on demand.



STREAM OPERATIONS – I

Java 8+

Intermediate Operations

- filter
- map
- mapToInt
- mapToLong
- mapToDouble
- flatMap
- flatMapToInt
- flatMapToLong
- flatMapToDouble
- sorted
- distinct
- limit
- skip
- peek

Terminal Operations

- forEach
- forEachOrdered
- collect
- reduce
- max
- min
- count
- toArray
- allMatch
- anyMatch
- noneMatch
- findFirst
- findAny

#CodingWalaShree

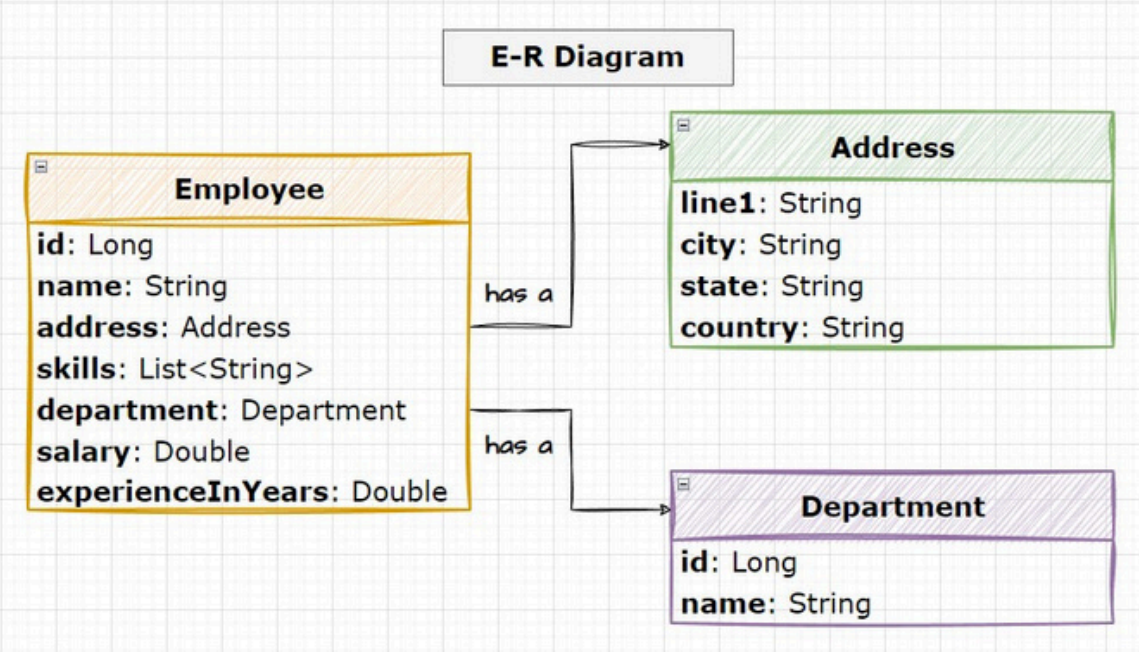
STREAM OPERATIONS – I

Java 8+

Project setup for Stream operations videos

Model Classes:

1. Employee
2. Department
3. Address



Repository Layer:

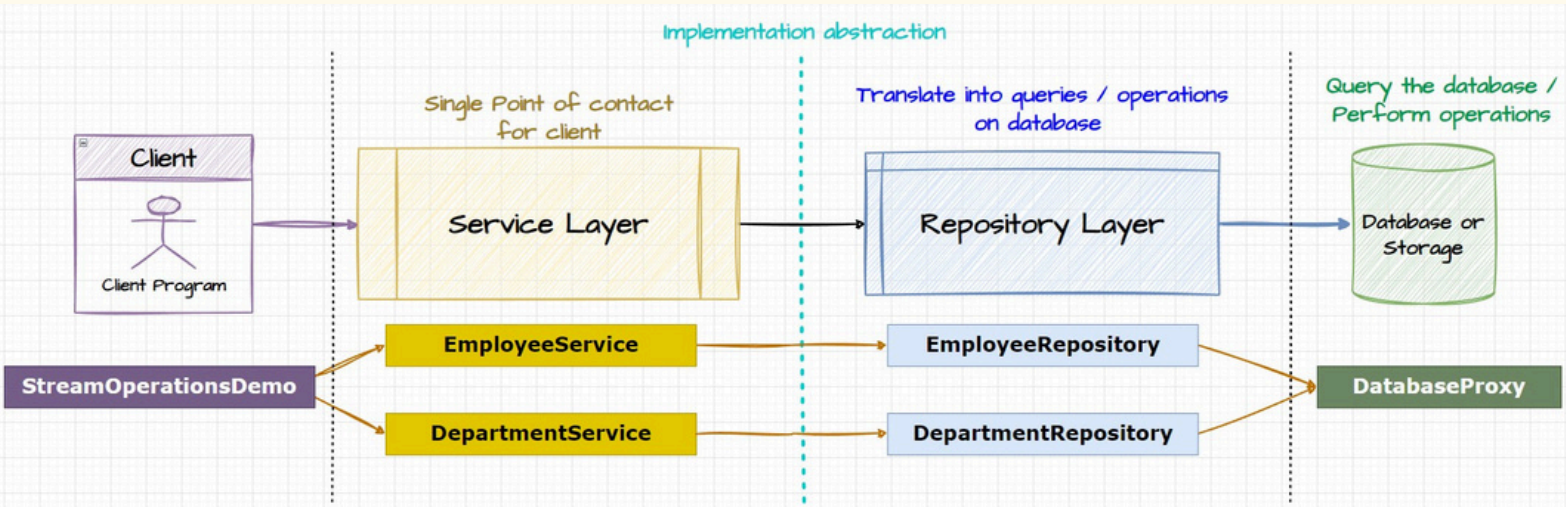
1. EmployeeRepository
2. DepartmentRepository

Service Layer:

1. EmployeeService
2. DepartmentService

Database: DatabaseProxy class consists of static data

Project Structure:



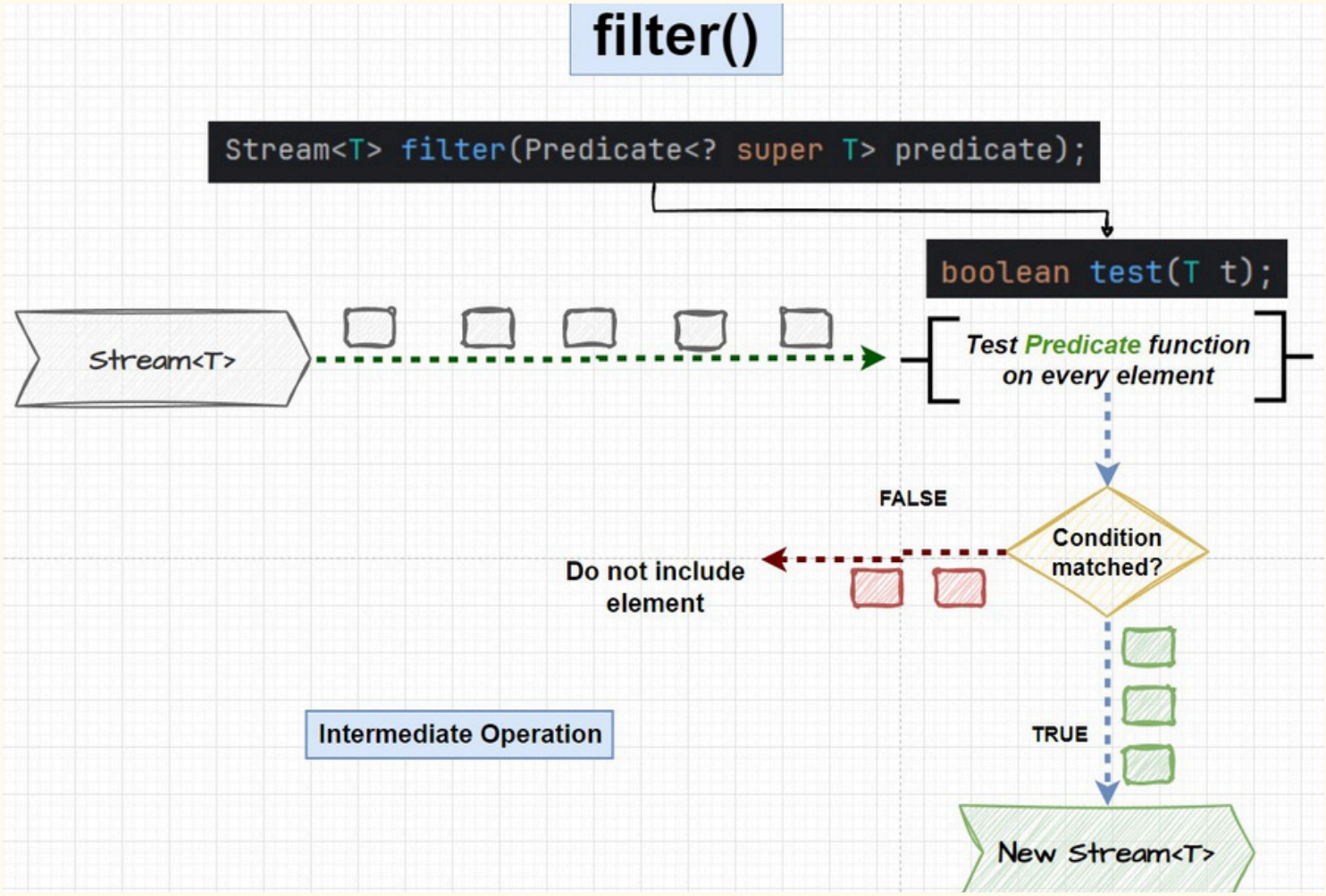
#CodingWalaShree

STREAM OPERATIONS – I

Java 8+

filter()

- An intermediate operation that returns a new stream consisting of the elements of this stream that match the given predicate.



Example 1: *Get list of employees whose name contains given sub-string*

```
public List<Employee> findByNameContains(String text) {
    return DatabaseProxy.getEmployees() List<Employee>
            .stream() Stream<...>
            .filter(e -> e.getName().contains(text))
            .collect(Collectors.toList());
}
```

#CodingWalaShree

STREAM OPERATIONS – I

Java 8+

filter()

-- [cntd.]

Example 2: *Get list of employees living in given city - Repository Layer*

```
public List<Employee> findByCity(String city) {  1 usage   codingv
    return DatabaseProxy.getEmployees() List<Employee>
            .stream() Stream<Employee>
            .filter(e -> e.getAddress().getCity().equals(city))
            .collect(Collectors.toList());
}
```

Example 3: *Get list of employees in given department having salary greater than given salary - Repository Layer*

```
public List<Employee> findByDepartmentIdAndSalaryGreaterThan 1
                        (long deptId, double minSalary) {
    return DatabaseProxy.getEmployees() List<Employee>
            .stream() Stream<Employee>
            .filter(e -> e.getDepartment().getId() == deptId
                        && e.getSalary() > minSalary)
            .collect(Collectors.toList());
}
```

STREAM OPERATIONS – I

Java 8+

filter()

-- [cntd.]

Example 4: *Get list of employees having given skill - Repository Layer*

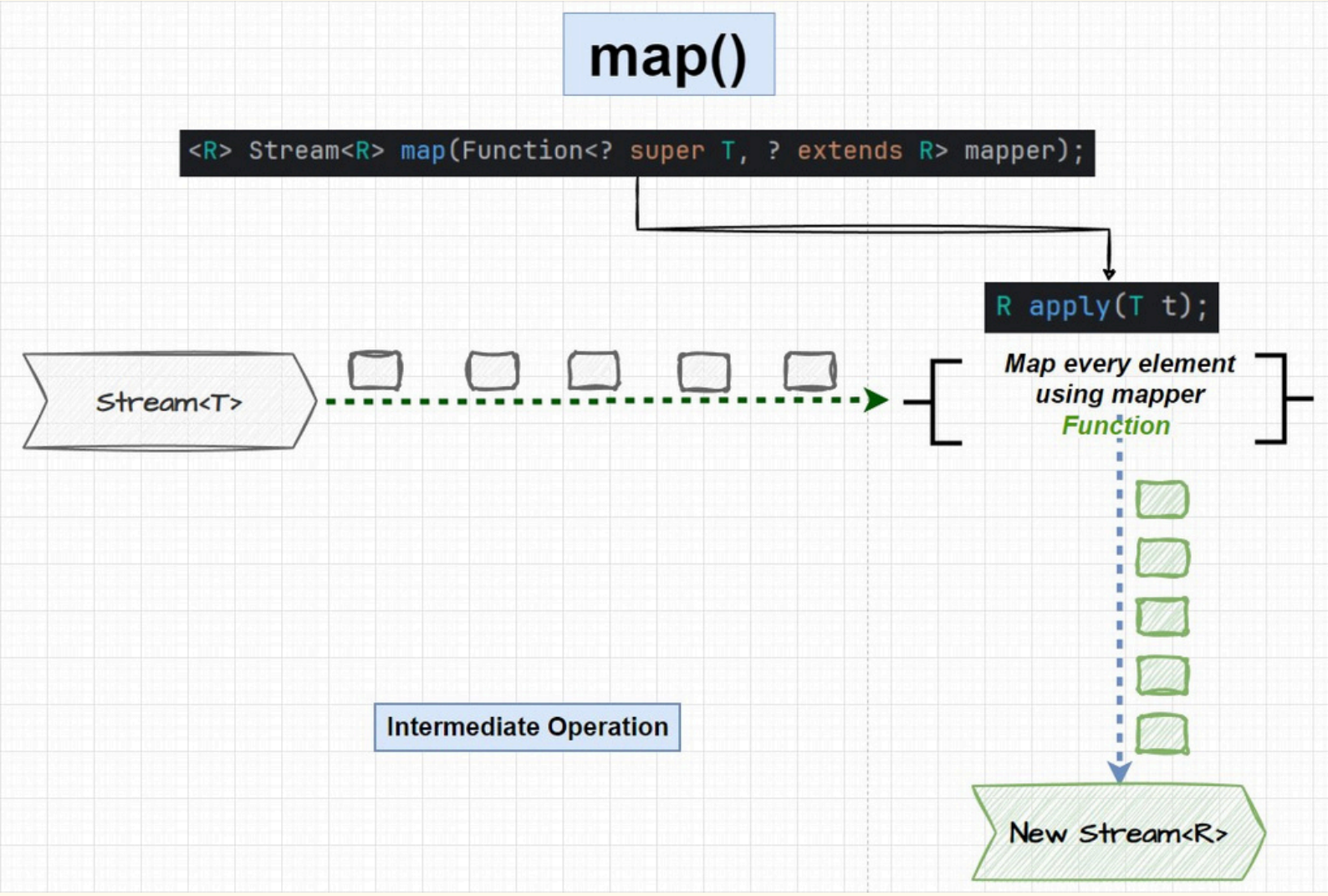
```
public List<Employee> findBySkill(String skill) {  1 usag
    return DatabaseProxy.getEmployees() List<Employee>
            .stream() Stream<Employee>
            .filter(e -> e.getSkills().contains(skill))
            .collect(Collectors.toList());
}
```

STREAM OPERATIONS – I

Java 8+

map()

- Returns a new stream consisting of the results of applying the given function to the elements of this stream.
- This is an intermediate operation.



#CodingWalaShree

STREAM OPERATIONS – I

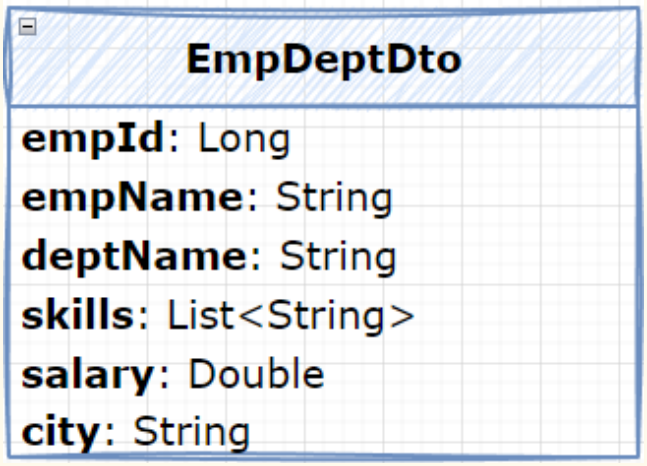
Java 8+

map()

-- [cntd.]

We additionally have a **EmpDeptDto** class which represents a DTO or Data Transfer Object.

DTO is a simple Java class used to transfer data between different layers of an application.



Example 1: *Get employee DTOs having given skill - Service Layer*

```
public List<EmpDeptDto> getEmployeesHavingSkill(String skill) {  1 usage
    return employeeRepository.findBySkill(skill) List<Employee>
            .stream() Stream<Employee>
            .map(EmpDeptDto::new) // passing constructor reference
            //.map(e -> new EmpDeptDto(e))  // explicitly creating with new
            .collect(Collectors.toList());
}
```

STREAM OPERATIONS – I

Java 8+

map()

-- [cntd.]

Example 2: *Get list of employee DTOs in given department having salary greater than given salary - Service Layer*

```
public List<EmpDeptDto> getHighlyPaidEmployees(long deptId, double minSalary) {
    return employeeRepository
            .findByDepartmentIdAndSalaryGreaterThan(deptId, minSalary) List<Employ
            .stream() Stream<Employee>
            .map(EmpDeptDto::new) Stream<EmpDeptDto>
            .collect(Collectors.toList());
}
```

STREAM OPERATIONS – I

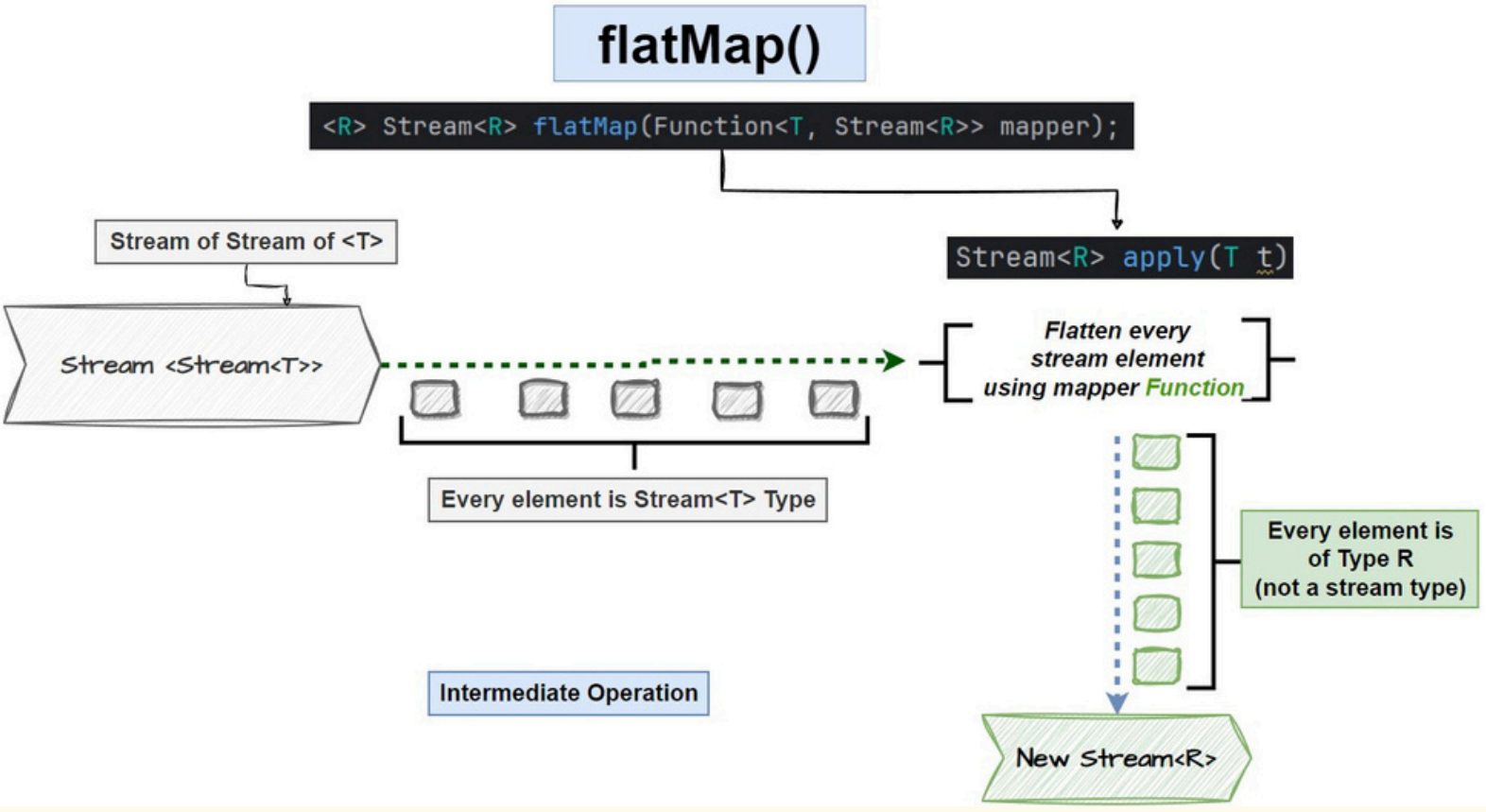
Java 8+

flatMap()

- An intermediate operation that 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.

- **What is flattening?:** In Java Streams, flattening refers to the process of converting a nested structure (like a list of lists) into a single-level structure using flatMap()

- When working with collections like **List<List<T>>**, applying **map()** would still return a **List<List<T>>**, which isn't always useful. **flatMap()** helps by merging inner lists into a single stream of elements.



#CodingWalaShree

STREAM OPERATIONS – I

Java 8+

flatMap() -- [cntd.]

Example 1: *Get all skills employees in a given department have - Service Layer*

```
public List<String> getSkillsByDeptId(long deptId) {
    return employeeRepository.findSkillsByDeptId(deptId)
            .stream()
            .flatMap(e -> e.getSkills().stream())
            .distinct()
            .collect(Collectors.toList());
}
```

Examples in this presentation are covered in my YouTube Video on Stream API Part 3🚀

🚀 Master Java Streams: filter(), map(), and flatMap() with Real-Project Examples!

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