PART 1: LAMBDA EXPRESSIONS (FULL DEEP MASTER CLASS)

STEP 1: Java 8 se pehle problem kya thi?

Java 7 tak code:

- ❌ Bahut **lamba**
- ❌ Bahut **boilerplate**
- ❌ Har chhoti cheez ke liye:
  - Interface
  - Class
  - Override
  - Object

Example: Java 7 me Thread

```
class MyThread implements Runnable {
    public void run() {
        System.out.println("Thread running");
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        Thread t = new Thread(t1);
        t.start();
    }
}
```

👉 Sirf ek line print karne ke liye **8–10 lines ka code** 😤

STEP 2: Lambda Expression kya hota hai?

✅ **Lambda = function ko short form me likhne ka tarika**

Java me pehle:

- Function ko likhne ke liye → class + interface chahiye

Java 8 me:

- Direct likh sakte ho → **Lambda**

☑ Definition (Interview ready line):

Lambda Expression is a short way to implement a Functional Interface using anonymous function.

STEP 3: Lambda ka GENERAL SYNTAX

```
(parameter) -> { body }
```

Examples:

Normal Method	Lambda
`void show()`	`() -> System.out.println("Hello")`
`int add(int a,int b)`	`(a,b) -> a+b`

STEP 4: Functional Interface kya hota hai? (VERY IMPORTANT)

✅ **Functional Interface = Jisme sirf 1 abstract method hota hai**

☑ Example:

```
interface A {
    void show();   // ✅ Only one method
}
```

❌ Galat:

```
interface A {
    void show();
    void display();   // ❌ 2 abstract methods → Lambda NOT allowed
}
```

👉 **Rule yaad rakho:**

❌ No Functional Interface = ❌ No Lambda

STEP 5: Lambda ke BASIC TYPES (One by One)

(A) No Parameter Lambda

```
interface A {
    void show();
}

public class Test {
    public static void main(String[] args) {
        A obj = () -> {
            System.out.println("Hello Java 8");
        };
        obj.show();
    }
}
```

Short form:

```
A obj = () -> System.out.println("Hello Java 8");
```

(B) One Parameter Lambda

```
interface A {
    void print(int a);
}

public class Test {
    public static void main(String[] args) {
        A obj = (a) -> System.out.println(a);
        obj.print(10);
    }
}
```

(C) Two Parameter Lambda + Return

```
interface A {
    int add(int a, int b);
}

public class Test {
    public static void main(String[] args) {
        A obj = (a, b) -> {
            return a + b;
        };
        System.out.println(obj.add(10, 20));
    }
}
```

Shortest form:

```
A obj = (a, b) -> a + b;
```

STEP 6: Lambda with REAL JAVA USE CASES

(1) Lambda with THREAD (Most Important)

❌ Old:

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Thread running");
    }
};
```

```
Thread t = new Thread(r);
t.start();
```

✅ Java 8:

```
Runnable r = () -> System.out.println("Thread running");
Thread t = new Thread(r);
t.start();
```

👉 **Multithreading + Java 8 = Direct Interview Question**

(2) Lambda with COLLECTION

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.forEach(i -> System.out.println(i));
```

(3) Lambda with COMPARATOR

```
Collections.sort(list, (a, b) -> a - b);
```

STEP 7: Lambda ke IMPORTANT RULES (Interview Golden Points)

✅ Lambda:

- Functional Interface ke saath kaam karta hai
- Code short banata hai
- Readability badhata hai

❌ Lambda:

- Constructor nahi hota
- Instance variable nahi hota
- Multi-method interface pe kaam nahi karta

FIRST PRACTICE SET

Q1:

Lambda se thread banao jo print kare:
`"Database task running"`

Q2:

Lambda se do numbers ka:

- Addition

- Subtraction
- Multiplication

Q3:
ArrayList banao:
`5, 15, 25`

Lambda se print karo.

Q4:
Lambda se maximum nikaalo:
`45, 78`

Q5 (INTERVIEW):
Explain in your words:
Lambda kya hota hai aur kyun use karte hain?

PART 2: FUNCTIONAL INTERFACES (FULL DEEP MASTER CLASS)

Hum is part ko **5 solid steps** me cover karenge:

1️⃣ Functional Interface kya hota hai
2️⃣ `@FunctionalInterface` annotation
3️⃣ Built-in Functional Interfaces:
- Predicate
- Function
- Consumer
- Supplier

4️⃣ Lambda + Inka real use
5️⃣ Interview Questions

STEP 1: Functional Interface kya hota hai?

Functional Interface = Aisa interface jisme sirf ek hi abstract method ho.

☑ Example:

```
interface A {
    void show();   // ✅ Only one abstract method
}
```

Isliye hum iske saath lambda likh sakte hain:

```
A obj = () -> System.out.println("Hello");
```

❌ Ye Functional Interface nahi hai:

```
interface A {
    void show();
    void display();   // ❌ 2 abstract methods
}
```

👉 Iske saath lambda **KAAM NAHI karega**.

STEP 2: `@FunctionalInterface` Annotation kya hota hai?

Ye annotation compiler ko batata hai:

✅ "Ye interface sirf 1 abstract method ka hona chahiye."

☑️ Example:

```
@FunctionalInterface
interface A {
    void show();
}
```

❌ Agar galti se 2 method likh do:

```
@FunctionalInterface
interface A {
    void show();
    void display();  // ❌ Compile-time error
}
```

👉 **Advantage:**

Tum galti se rule break nahi kar paoge ✅

AB AATA HAI JAVA 8 KA SABSE IMPORTANT PART

Built-in Functional Interfaces (java.util.function package)

Java ne khud **4 sabse powerful functional interfaces** bana rakhe hain:

Interface	Kaam
Predicate	Condition check
Function	Input → Output
Consumer	Sirf input, no return
Supplier	Sirf output, no input

1️⃣ PREDICATE (Test / Condition ke liye)

☑️ Definition:

Predicate input leta hai aur **boolean return karta hai**.

☑️ Method:

```
boolean test(T t);
```

☑️ Example 1: Even / Odd check

```
import java.util.function.Predicate;

public class Test {
    public static void main(String[] args) {

        Predicate<Integer> p = (n) -> n % 2 == 0;

        System.out.println(p.test(10)); // true
        System.out.println(p.test(7));  // false
    }
}
```

☑️ Example 2: String length check

```
Predicate<String> p = s -> s.length() > 5;

System.out.println(p.test("Java"));       // false
System.out.println(p.test("Programming")); // true
```

2️⃣ FUNCTION (Input → Output conversion)

☑️ Definition:

Function input bhi leta hai aur **output bhi return karta hai**.

☑️ Method:

```
R apply(T t);
```

☑️ Example 1: Square nikalna

```
import java.util.function.Function;

public class Test {
    public static void main(String[] args) {
```

```
        Function<Integer, Integer> f = n -> n * n;

        System.out.println(f.apply(5));  // 25
    }
}
```

☑ Example 2: String ko uppercase me convert karna

```
Function<String, String> f = s -> s.toUpperCase();

System.out.println(f.apply("java")); // JAVA
```

3 CONSUMER (Sirf input, koi return nahi)

☑ Definition:

Consumer sirf value **consume karta hai**, kuch return nahi karta.

☑ Method:

```
void accept(T t);
```

☑ Example 1: Print value

```
import java.util.function.Consumer;

public class Test {
    public static void main(String[] args) {

        Consumer<String> c = s -> System.out.println(s);

        c.accept("Hello Java 8");
    }
}
```

☑ Example 2: List ke sab elements print

```
import java.util.*;

list.forEach(i -> System.out.println(i));
```

`forEach()` internally **Consumer** use karta hai ✅

4️⃣ SUPPLIER (Sirf output, koi input nahi)

☑️ Definition:

Supplier **kuch generate karta hai**, koi input nahi leta.

☑️ Method:

```
T get();
```

☑️ Example 1: Random number generate

```
import java.util.function.Supplier;
import java.util.Random;

public class Test {
    public static void main(String[] args) {

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

        System.out.println(s.get());
    }
}
```

☑️ Example 2: Fixed message supply

```
Supplier<String> s = () -> "Welcome to Java 8";

System.out.println(s.get());
```

STEP 3: Predicate + Function + Consumer + Supplier ek saath

```
Predicate<Integer> p = n -> n > 10;
Function<Integer, Integer> f = n -> n * 2;
Consumer<Integer> c = n -> System.out.println(n);
Supplier<Integer> s = () -> 50;

int x = s.get();    // 50

if (p.test(x)) {   // true
    int y = f.apply(x); // 100
    c.accept(y);  // print 100
```

```
}
```

STEP 4: Interview IMPORTANT Points

✅ Predicate → boolean
✅ Function → input → output
✅ Consumer → input only
✅ Supplier → output only
✅ Ye sab:

- Stream API
- Spring Boot
- Data Processing

me heavily use hote hain ✅

PRACTICE (VERY IMPORTANT – YE KARNA HI HAI)

Q1:

Predicate use karke check karo:
`Number 50 > 30 ?`

Q2:

Function use karke:
`Number ka cube (n³)`

nikaalo.

Q3:

Consumer use karke:
`"Java Backend Developer"`

print karao.

Q4:

Supplier use karke:
`OTP ya koi random number generate karo.`

Q5 (INTERVIEW):
Difference explain karo:
`Predicate vs Function vs Consumer vs Supplier`

PART 3: STREAM API – FULL DEEP MASTER CLASS

Hum is topic ko **6 solid steps me cover karenge:**

1️⃣ Stream kya hota hai & Collection se difference
2️⃣ Stream ka flow (Source → Intermediate → Terminal)
3️⃣ filter()
4️⃣ map()
5️⃣ reduce()
6️⃣ collect(), forEach(), sorted(), count()
7️⃣ Real-world examples
8️⃣ Interview questions

STEP 1: Stream API kya hota hai?

✅ **Stream ek data processing pipeline hai jo Collection ke data par fast, clean aur functional tarike se kaam karta hai.**

Simple words me:

- Collection = data store karta hai
- Stream = data ko **process** karta hai

❌ Collection (Old Style)

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);

for (int i : list) {
    if (i > 20) {
        System.out.println(i);
    }
}
```

☑️Stream (Java 8 Style)

```
list.stream()
    .filter(i -> i > 20)
```

```
    .forEach(i -> System.out.println(i));
```

👉 **Same kaam, 60% kam code, zyada readable** ✅

STEP 2: Stream ka FLOW (VERY IMPORTANT)

Har stream 3 parts me kaam karta hai:
`SOURCE → INTERMEDIATE → TERMINAL`

Example:

```
list.stream()                    // Source
    .filter(i -> i > 20)         // Intermediate
    .forEach(i -> System.out.println(i)); // Terminal
```

Rules:

✅ Source = Collection / Array
✅ Intermediate = filter(), map(), sorted()
✅ Terminal = forEach(), collect(), reduce(), count()
❌ Terminal ke baad stream dubara use nahi hota

STEP 3: filter() – Data ko condition se chhanna

✅ filter() Predicate use karta hai
Input → boolean output

☑ Example 1: Even numbers filter karo

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(15);
list.add(20);
list.add(25);

list.stream()
    .filter(n -> n % 2 == 0)
    .forEach(n -> System.out.println(n));
// Output: 10, 20
```

☑ Example 2: Names filter karo (length > 4)

```
ArrayList<String> names = new ArrayList<>();
names.add("Aman");
```

```
names.add("Rahul");
names.add("Suresh");
names.add("Amit");

names.stream()
     .filter(name -> name.length() > 4)
     .forEach(name -> System.out.println(name));
// Rahul, Suresh
```

STEP 4: map() – Data ko transform karta hai

✅ map() Function use karta hai
Input → Output

☑ Example 1: Numbers ka square

```
ArrayList<Integer> list = new ArrayList<>();
list.add(2);
list.add(3);
list.add(4);

list.stream()
    .map(n -> n * n)
    .forEach(n -> System.out.println(n));
// Output: 4, 9, 16
```

☑ Example 2: Names ko uppercase me convert karo

```
names.stream()
     .map(name -> name.toUpperCase())
     .forEach(name -> System.out.println(name));
```

STEP 5: filter() + map() combo (REAL INTERVIEW TYPE)

☑ Task:

Even numbers lo aur unka square nikaalo

```
list.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .forEach(n -> System.out.println(n));
```

STEP 6: reduce() – Final result banata hai

✅ reduce() sab elements ko **ek single value me convert karta hai**

☑ Example: Sum of all numbers

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);

int sum = list.stream()
               .reduce(0, (a, b) -> a + b);

System.out.println(sum); // 60
```

👉 0 = initial value

👉 (a, b) -> a + b = lambda

STEP 7: collect() – Stream ko dubara Collection me badalna

☑ Example: Even numbers ko nayi List me store karo

```
List<Integer> evenList =
list.stream()
    .filter(n -> n % 2 == 0)
    .collect(java.util.stream.Collectors.toList());

System.out.println(evenList);
```

STEP 8: sorted(), count(), forEach()

sorted()

```
list.stream().sorted().forEach(System.out::println);
```

count()

```
long total = list.stream().count();
System.out.println(total);
```

forEach()

```
list.forEach(n -> System.out.println(n));
```

STREAM API REAL-WORLD EXAMPLE (PROJECT TYPE)

Employee Names Filter + Uppercase

```
ArrayList<String> empNames = new ArrayList<>();
empNames.add("Aman");
empNames.add("Ravi");
empNames.add("Suresh");
empNames.add("Ankit");

empNames.stream()
        .filter(name -> name.startsWith("A"))
        .map(name -> name.toUpperCase())
        .forEach(name -> System.out.println(name));
```

STEP 9: IMPORTANT INTERVIEW QUESTIONS

✅ Difference:

- Collection vs Stream
  - ✅ filter() vs map()
  - ✅ map() vs flatMap()
  - ✅ Intermediate vs Terminal operations
  - ✅ reduce() use case
  - ✅ Why Stream is lazy?

PRACTICE (YE MUST SOLVE KARNA HAI)

Q1:

List:

```
10, 15, 20, 25, 30
```

➞ Stream se sirf **odd numbers print karo**

Q2:
Same list ka:
➞ Stream se **square nikalo**

Q3:
Names list:
`ram, shyam, mohan, sohan`

➞ Sirf **"m" se start wale names print karo**

Q4:
Numbers ka:
➞ Stream se **sum nikalo**

Q5 (INTERVIEW):
Explain in your words:
`Stream API kya hota hai aur kyun use hota hai?`

PART 4: METHOD REFERENCES (FULL DEEP MASTER CLASS)

Hum is topic ko **5 practical steps** me cover karenge:

1️⃣ Method Reference kya hota hai
2️⃣ Types of Method References
3️⃣ Static Method Reference
4️⃣ Instance Method Reference
5️⃣ Constructor Reference
6️⃣ Lambda vs Method Reference
7️⃣ Interview Questions

STEP 1: Method Reference kya hota hai?

✅ **Method Reference = Lambda ka short & clean version jo directly existing method ko refer karta hai.**

Simple words me:
Agar tumhara lambda:
`(x) -> someMethod(x)`

Toh usko tum is tarah bhi likh sakte ho:
`ClassName::someMethod`

👉 Isse:
- Code aur **chhota**
- **Readable**
- **Professional** ban jaata hai ✅

STEP 2: Types of Method References (3 Types)

Type	Syntax	Example
Static	`ClassName::methodName`	`Math::sqrt`
Instance (object)	`object::methodName`	`str::toUpperCase`
Constructor	`ClassName::new`	`Student::new`

TYPE 1: STATIC METHOD REFERENCE

❌ Without Method Reference (Lambda)
```
Function<Integer, Double> f = n -> Math.sqrt(n);
System.out.println(f.apply(25));
```

☑ With Method Reference
```
Function<Integer, Double> f = Math::sqrt;
System.out.println(f.apply(25)); // 5.0
```

👉 yahan:
- `Math.sqrt(n)` → `Math::sqrt`

☑ Another Static Example
```
Function<String, Integer> f = Integer::parseInt;
System.out.println(f.apply("100")); // 100
```

TYPE 2: INSTANCE METHOD REFERENCE (Object wala)

❌ With Lambda

```
Function<String, String> f = s -> s.toUpperCase();
System.out.println(f.apply("java"));
```

☑ With Method Reference

```
Function<String, String> f = String::toUpperCase;
System.out.println(f.apply("java")); // JAVA
```

👉 `s -> s.toUpperCase()` → `String::toUpperCase`

☑ Instance method with real object

```
class MyPrinter {
    void print(String s) {
        System.out.println(s);
    }
}

public class Test {
    public static void main(String[] args) {
        MyPrinter p = new MyPrinter();

        Consumer<String> c = p::print;
        c.accept("Hello Java 8");
    }
}
```

TYPE 3: CONSTRUCTOR REFERENCE (Most Advanced)

❌ With Lambda

```
Supplier<Student> s = () -> new Student();
```

☑ With Constructor Reference

```
Supplier<Student> s = Student::new;
```

☑ Example Fully Working

```
class Student {
    Student() {
        System.out.println("Student object created");
    }
}

public class Test {
    public static void main(String[] args) {
        Supplier<Student> s = Student::new;
        s.get();   // Object create hoga
    }
}
```

STEP 3: Lambda vs Method Reference

Feature	Lambda	Method Reference
Code Size	Thoda bada	Aur chhota
Readability	Good	Very clean
Logic	Khud likhte ho	Existing method reuse
Preference	Jab custom logic ho	Jab method already exist kare

✅ **Rule:**

Agar existing method use ho sakta hai → Method Reference best ✅

STEP 4: Method Reference with Streams (REAL PROJECT USE)

☑ List print

```
list.forEach(System.out::println);
```

☑ Sorting with Method Reference

```
Collections.sort(list, Integer::compareTo);
```

STEP 5: IMPORTANT INTERVIEW QUESTIONS

✅ Lambda vs Method Reference

✅ Types of Method Reference

✅ Constructor Reference use
✅ Where do we use `ClassName::methodName`?
✅ Can every lambda be converted to a method reference?
➡️ ❌ NO — only when logic kisi existing method me ho.

PRACTICE (VERY IMPORTANT)

Q1:
Use Method Reference to:
`Print all elements of List`

Q2:
Use Method Reference to:
`Convert string to Integer`

Q3:
Use Constructor Reference to:
`Create object of any class`

Q4 (Interview):
Explain in your words:
`Lambda vs Method Reference difference`

PART 5: OPTIONAL CLASS (FULL DEEP MASTER CLASS)

Hum is topic ko **7 clear steps** me cover karenge:
1️⃣ NullPointerException problem kya hai
2️⃣ Optional kya hota hai
3️⃣ Optional object kaise banate hain
4️⃣ Data kaise nikalte hain (get, orElse, etc.)
5️⃣ ifPresent()
6️⃣ Real Project Example
7️⃣ Interview Questions + Practice

STEP 1: NullPointerException PROBLEM

Tum ye problem to bahut baar dekhi hogi 👇

```
String name = null;
System.out.println(name.length()); // 👿 Exception
```

❌ Output:
```
NullPointerException
```

✅ Reason:
- Tum `null` par method call kar rahe ho.

Old Solution (Java 7 style)
```
if(name != null){
    System.out.println(name.length());
}
```

👉 Lekin har jagah `if != null` lagana **dirty coding** hoti hai ❌

STEP 2: OPTIONAL KYA HOTA HAI?

Optional ek wrapper class hai jo value bhi hold kar sakta hai aur empty bhi ho sakta hai.

Simple definition:

✓ Value ho sakti hai

✓ Value missing bhi ho sakti hai

✓ NullPointerException se safely bachaata hai ✅

STEP 3: OPTIONAL OBJECT KAISE BANATE HAIN?

1. of() → Jab value CERTAIN ho
```
Optional<String> op = Optional.of("Java");
```

⚠️ Agar `null` diya → Exception aayega

2. ofNullable() → Jab value null ho sakti ho
```
Optional<String> op = Optional.ofNullable(null);
```

✅ Yeh **safe hai**

3. empty() → Bilkul empty object
```
Optional<String> op = Optional.empty();
```

STEP 4: DATA KAISE NIKALEIN?

❌ get() (Direct, risky)

```
Optional<String> op = Optional.of("Hello");
System.out.println(op.get()); // Hello
```

⚠️ Agar empty ho → Exception ❌

☑ orElse() (Safe + Professional)

```
Optional<String> op = Optional.ofNullable(null);
System.out.println(op.orElse("Default Value"));
```

✅ Output:

```
Default Value
```

☑ orElseGet() (Lazy execution)

```
System.out.println(op.orElseGet(() -> "Generated Value"));
```

☑ orElseThrow()

```
op.orElseThrow(() -> new RuntimeException("Value missing"));
```

STEP 5: ifPresent() (Without if condition)

❌ Old Style

```
if(name != null){
   System.out.println(name);
}
```

☑ Java 8 Style

```
Optional<String> op = Optional.of("Java");

op.ifPresent(x -> System.out.println(x));
```

🔥 Clean, short, professional!

STEP 6: REAL PROJECT EXAMPLE (SPRING BOOT TYPE)

WITHOUT Optional (Risky)

```
User user = userRepo.findById(1);
System.out.println(user.getName()); // NPE possible
```

WITH Optional (Professional)

```
Optional<User> user = userRepo.findById(1);

user.ifPresent(u -> System.out.println(u.getName()));
```

OR

```
User u = userRepo.findById(1)
        .orElseThrow(() -> new RuntimeException("User not found"));
```

यही industry standard hai 🔥

STEP 7: IMPORTANT INTERVIEW QUESTIONS

- ✅ Optional kya hota hai?
- ✅ of() vs ofNullable() difference
- ✅ orElse vs orElseGet
- ✅ Can Optional store null? → ❌ NO
- ✅ Why Optional is used in Spring Data JPA?

OPTIONAL SUMMARY

Method	Use
of()	Jab sure ho value null nahi
ofNullable()	Jab value null ho sakti
empty()	Jab koi value hi nahi
get()	Direct value (risky)
orElse()	Default value
orElseThrow()	Custom exception
ifPresent()	If value present then use

PRACTICE QUESTIONS (MUST DO)

Q1:
Optional<Integer> banakar value print karo

Q2:
Optional<String> me null store karo aur orElse use karo

Q3:
orElseThrow ka example likho

Q4 (Interview):
Optional NullPointerException ko kaise avoid karta hai?

PART 6: JAVA 8 + COLLECTIONS (REAL-WORLD MASTER USAGE)

Is part me hum seekhenge:

1️⃣ List + Streams
2️⃣ Filter, Map, Sort on Collection
3️⃣ Object List par Java 8
4️⃣ Grouping using Map
5️⃣ forEach + Lambda
6️⃣ Interview Level Patterns
7️⃣ Mini Real-World Example

1️⃣ List + Streams (Foundation)

```
List<Integer> list = Arrays.asList(10, 20, 30, 40, 50);

list.stream()
    .forEach(x -> System.out.println(x));
```

✅ Stream = collection par functional way me kaam karna
✅ forEach = loop ka replacement

2️⃣ FILTER (Data Chhantna)

EVEN numbers sirf print karo:

```
list.stream()
    .filter(x -> x % 2 == 0)
    .forEach(x -> System.out.println(x));
```

✅ filter = condition lagata hai

3️⃣ MAP (Data Transform karna)

Sab numbers ka square:

```
list.stream()
    .map(x -> x * x)
    .forEach(x -> System.out.println(x));
```

✅ map = data ko badalta hai

4️⃣ SORT (Ascending / Descending)

```
list.stream()
    .sorted()
    .forEach(x -> System.out.println(x));
```

Descending:

```
list.stream()
    .sorted((a, b) -> b - a)
    .forEach(x -> System.out.println(x));
```

✅ Comparator + Lambda ka real use 🔥

5️⃣ OBJECT LIST + JAVA 8 (VERY IMPORTANT)

🔹 Employee Class

```
class Employee {
    int id;
    String name;
    int salary;
    Employee(int id, String name, int salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
```

```
    }

    public String toString() {
        return id + " " + name + " " + salary;
    }
}
```

🔹 List of Employees

```
List<Employee> empList = new ArrayList<>();

empList.add(new Employee(1, "Amit", 30000));
empList.add(new Employee(2, "Rahul", 50000));
empList.add(new Employee(3, "Neha", 40000));
```

☑️ Salary > 40000 wale employees print karo:

```
empList.stream()
       .filter(e -> e.salary > 40000)
       .forEach(e -> System.out.println(e));
```

☑️ Sirf naam print karo:

```
empList.stream()
       .map(e -> e.name)
       .forEach(n -> System.out.println(n));
```

6️⃣ GROUPING (Map + Streams)

Salary ke type se group karo:

```
empList.stream()
       .collect(Collectors.groupingBy(e -> e.salary))
       .forEach((k, v) -> System.out.println(k + " " + v));
```

✅ Ye **Spring Boot reporting APIs** me direct use hota hai 🔥

7️⃣ forEach + Lambda (Clean Code)

```
empList.forEach(e -> System.out.println(e.name));
```

✅ Loop ka modern version ✅

8️⃣ INTERVIEW LEVEL QUESTIONS

✅ filter vs map
✅ stream vs collection
✅ why Java 8 introduced
✅ groupingBy use
✅ lambda vs anonymous class
✅ Optional + Stream relation

9️⃣ MINI REAL-WORLD CASE (PROJECT LOGIC)

Task:
40,000 se zyada salary wale employees ke naam uppercase me print karo.

```
empList.stream()
       .filter(e -> e.salary > 40000)
       .map(e -> e.name.toUpperCase())
       .forEach(System.out::println);
```

✅ 1 line me full business logic 🔥
✅ Yahi Spring Boot service layer me hota hai ✅