# JDK 1.8 Features

***-Scribbles by Miss. Geetanjali Gajbhar***

- **Lambda Expressions** – Enables functional programming with concise syntax.
- **Functional Interfaces** – Interfaces with a single abstract method (e.g., `Runnable`, `Comparator`).
- **Stream API** – Processes collections in a functional, declarative way (filter, map, reduce).
- **Default & Static Methods in Interfaces** – Allows method implementation inside interfaces.
- **java.time Package** – New Date & Time API (replaces `Date` and `Calendar`).
- **Optional Class** – Helps avoid `NullPointerException`.
- **Method References** – Shorthand for calling methods via `::` operator.
- **Collectors API** – Used to collect stream results (e.g., to list, set, map).
- **Nashorn JavaScript Engine** – Executes JavaScript code from Java (deprecated later).

## Lambda Expressions in Java 8

## Definition:

A **lambda expression** is a short block of code that takes in parameters and returns a value. It lets you write **anonymous functions** (functions without names) in a more concise and readable way.

---

## Syntax:

```
(parameter1, parameter2, ...) -> { body }
```

---

## Basic Example:

```
(int a, int b) -> {
    return a + b;
}
```

Or simplified:

```
(a, b) -> a + b
```

---

## Key Components:

- **Parameters**: Like method parameters.
- **Arrow token (`->`)**: Separates parameters from the body.
- **Body**: Can be a single expression or a block of code.

---

## Why Use Lambda?

- Reduces **boilerplate code**.
- Enables **functional programming**.
- Enhances code **readability and maintainability**.
- Works best with **functional interfaces** (interfaces with exactly one abstract method).

## Functional Interface Example:

```
@FunctionalInterface
interface MyOperation {
    int operate(int a, int b);
}
```

## Lambda Implementation:

```
MyOperation add = (a, b) -> a + b;
System.out.println(add.operate(5, 3)); // Output: 8
```

## Common Functional Interfaces in `java.util.function` package:

| Interface | Method | Description |
|---|---|---|
| `Predicate<T>` | `test(T t)` | Returns boolean |
| `Function<T,R>` | `apply(T t)` | Converts T to R |
| `Consumer<T>` | `accept(T t)` | Takes T, returns nothing |
| `Supplier<T>` | `get()` | Takes nothing, returns T |

## Examples in Action:

1. **Runnable using Lambda:**

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

1. **Comparator using Lambda:**

```
List<String> list = Arrays.asList("Apple", "Orange", "Banana");
Collections.sort(list, (s1, s2) → s1.compareTo(s2));
```

1. **Stream + Lambda:**

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);
nums.stream().filter(n → n % 2 == 0).forEach(System.out::println);
```

---

## When to Use:

- With **Streams**, **Collections**, and **Callbacks**
- For **short, throwaway implementations**
- In **event handling** (GUI, web, etc.)

---

## Limitations:

- Cannot use `this` to refer to lambda itself.
- No checked exceptions inside lambda unless handled.

**lambda expressions can only be used with functional interfaces** in Java.

---

## Why?

Because a lambda in Java is essentially a **syntactic shortcut** for creating an **implementation of a functional interface**—i.e., an interface with only **one abstract method**.

---

## What happens internally:

When you write:

```
Runnable r = () → System.out.println("Hello");
```

It's **internally converted** to:

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

---

## Cannot use with:

- Classes
- Interfaces with **more than one abstract method**
- Abstract classes
- Enums

---

## Can use with:

- Your own custom functional interfaces
- Predefined functional interfaces in `java.util.function`

- Any interface with a single abstract method—even if it has default/static methods

---

**Summary**: Lambda = Functional interface ONLY

## Basic Lambda Usage

### Example 1: No parameter, just print

```
Runnable r = () → System.out.println("Hello Lambda!");
r.run();
```

### Example 2: One parameter – square of a number

```
Function<Integer, Integer> square = x → x * x;
System.out.println(square.apply(5)); // Output: 25
```

### Example 3: Two parameters – addition

```
BiFunction<Integer, Integer, Integer> add = (a, b) → a + b;
System.out.println(add.apply(3, 7)); // Output: 10
```

---

## Use with Collections & Stream API

### Example 4: Filter even numbers from a list

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream()
```

```
        .filter(n -> n % 2 == 0)
        .forEach(System.out::println); // Output: 2 4 6
```

### Example 5: Sort list using lambda

```
List<String> names = Arrays.asList("Ravi", "Amit", "Zoya", "Geet");
Collections.sort(names, (a, b) -> a.compareTo(b));
System.out.println(names); // Sorted order
```

### Example 6: Use with Predicate – check string length > 5

```
Predicate<String> isLong = s -> s.length() > 5;
System.out.println(isLong.test("Geetanjali")); // true
```

---

## Custom Functional Interface & Complex Logic

### Example 7: Custom functional interface with multiple operations

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

public class LambdaTest {
    public static void main(String[] args) {
        MathOperation multiply = (a, b) -> a * b;
        MathOperation power = (a, b) -> (int)Math.pow(a, b);
```

```
        System.out.println("Multiply: " + multiply.operate(4, 5)); // 20
        System.out.println("Power: " + power.operate(2, 3));       // 8
    }
}
```

## Example 8: Chain operations using `Function` and `andThen()`

```
Function<Integer, Integer> doubleIt = n -> n * 2;
Function<Integer, Integer> squareIt = n -> n * n;

Function<Integer, Integer> chained = doubleIt.andThen(squareIt);
System.out.println(chained.apply(3)); // (3*2)=6, (6*6)=36
```

## Example 9: Grouping by string length using lambda + collectors

```
List<String> words = Arrays.asList("apple", "banana", "cat", "dog", "elephan
t");
Map<Integer, List<String>> grouped = words.stream()
    .collect(Collectors.groupingBy(s -> s.length()));

System.out.println(grouped);
// Output: {3=[cat, dog], 5=[apple], 6=[banana], 8=[elephant]}
```

---

## Predefined functional interface and their uses

Here are the **predefined functional interfaces** from `java.util.function` package with their **uses and examples**:

## 1. Predicate<T>

- **Method:** `boolean test(T t)`
- **Use:** To evaluate a condition (returns true/false)

```
Predicate<String> isLong = s → s.length() > 5;
System.out.println(isLong.test("Geet")); // false
```

## 2. Function<T, R>

- **Method:** `R apply(T t)`
- **Use:** To transform one value to another

```
Function<String, Integer> length = s → s.length();
System.out.println(length.apply("Geetanjali")); // 10
```

## 3. Consumer<T>

- **Method:** `void accept(T t)`
- **Use:** To perform an action without returning anything

```
Consumer<String> print = s → System.out.println("Hello, " + s);
print.accept("Students"); // Hello, Students
```

## 4. Supplier<T>

- **Method:** `T get()`

- **Use:** To return a value without any input

```
Supplier<Double> random = () → Math.random();
System.out.println(random.get());
```

---

## 5. BiPredicate<T, U>

- **Method:** `boolean test(T t, U u)`
- **Use:** Condition check with two inputs

```
BiPredicate<Integer, Integer> isGreater = (a, b) → a > b;
System.out.println(isGreater.test(10, 5)); // true
```

---

## 6. BiFunction<T, U, R>

- **Method:** `R apply(T t, U u)`
- **Use:** Convert two inputs to a single output

```
BiFunction<Integer, Integer, Integer> add = (a, b) → a + b;
System.out.println(add.apply(5, 3)); // 8
```

---

## 7. BiConsumer<T, U>

- **Method:** `void accept(T t, U u)`
- **Use:** Take two inputs and perform an action

```
BiConsumer<String, Integer> printAge = (name, age) →
    System.out.println(name + " is " + age + " years old");
```

```
printAge.accept("Geet", 25);
```

---

## 8. UnaryOperator<T> (extends `Function<T, T>`)

- **Use:** Operates on single input and returns same type

```
UnaryOperator<Integer> square = n -> n * n;
System.out.println(square.apply(4)); // 16
```

---

## 9. BinaryOperator<T> (extends `BiFunction<T, T, T>`)

- **Use:** Takes two same-type inputs, returns same type

```
BinaryOperator<Integer> multiply = (a, b) -> a * b;
System.out.println(multiply.apply(3, 4)); // 12
```

**practice examples** for each of the commonly used predefined functional interfaces from `java.util.function` —

---

## 1. Predicate<T> Practice

```
Predicate<String> startsWithG = s -> s.startsWith("G");
System.out.println(startsWithG.test("Geetanjali")); // true
System.out.println(startsWithG.test("Anjali"));     // false
```

---

## 2. Function<T, R> Practice

```
Function<String, Integer> wordLength = str -> str.length();
```

```
System.out.println(wordLength.apply("Lambda")); // 6
```

## 3. Consumer<T> Practice

```
Consumer<String> greet = name → System.out.println("Hello, " + name + "!");
greet.accept("Students"); // Hello, Students!
```

## 4. Supplier<T> Practice

```
Supplier<String> supplyName = () → "GeetCodeStudio";
System.out.println(supplyName.get()); // GeetCodeStudio
```

5. **BiPredicate<T, U> Practice**

```
BiPredicate<String, Integer> checkLength = (str, len) → str.length() == len;
System.out.println(checkLength.test("Java", 4)); // true
System.out.println(checkLength.test("Code", 5)); // false
```

## 6. BiFunction<T, U, R> Practice

```
BiFunction<String, String, String> fullName = (first, last) → first + " " + last;
System.out.println(fullName.apply("Geetanjali", "Gajbhar")); // Geetanjali Gajb
```

```
har
```

## 7. BiConsumer<T, U> Practice

```
BiConsumer<String, Integer> studentInfo = (name, score) →
    System.out.println(name + " scored " + score + " marks.");
studentInfo.accept("Ankit", 92); // Ankit scored 92 marks.
```

## 8. UnaryOperator<T> Practice

```
UnaryOperator<String> makeUpper = str → str.toUpperCase();
System.out.println(makeUpper.apply("lambda")); // LAMBDA
```

## 9. BinaryOperator<T> Practice

```
BinaryOperator<Integer> max = (a, b) → a > b ? a : b;
System.out.println(max.apply(20, 35)); // 35
```

## Functional Interface Basics

1. Create your own `@FunctionalInterface` named `Calculator` with one method `int operate(int a, int b)` and use lambda to implement addition, subtraction, and multiplication.
2. Write a lambda expression using `Predicate<Integer>` to filter out numbers that are both even and divisible by 5 from a list.

3. Use `Consumer<String>` to log messages with a timestamp prefix (e.g., `[2025-06-23 10:00] INFO: message`).

---

### Real-time Transformations & Filtering

1. Given a list of employees (name, salary), use `Function<Employee, String>` to extract names of employees earning more than ₹50,000.
2. Using `BiPredicate<String, String>`, check whether two email strings are equal ignoring case.
3. Implement a `Stream` pipeline with `map()`, `filter()`, and `collect()` to return a list of squared values of odd numbers.

---

## Stream & Lambda with Collections

1. From a `List<Order>`, filter orders with status `DELIVERED` and amount > ₹1000 using lambda and `Predicate`.
2. Write a lambda using `BinaryOperator<Integer>` to reduce a list of integers to their product.
3. Sort a list of `Product` objects by price using `Comparator` implemented with lambda.

---

### Chaining & Composition

1. Chain two `Function<String, String>` lambdas—one to trim a string, another to convert it to uppercase.
2. Use `Predicate<Employee>` and `.negate()` to filter out employees younger than 30.
3. Write a method that accepts a `Consumer<List<String>>` and inside the lambda, sort and print the list.

---

## Level 5: Custom Functional & Higher-Order Lambda Usage

1. Write a generic method `applyOperation(int a, int b, MathOperation op)` that uses lambda to perform different operations (add, subtract, divide).
2. Create a method that accepts a `Supplier<List<String>>` to generate dummy data for testing.

3. Use `BiConsumer<Map<String, Integer>, String>` to increase a given key's value by 10 in a map if it exists.
4. Write a lambda expression to print "Hello Java 8".
5. Create a `Predicate` to check if a number is even.
6. Use `Consumer` to display any given string in uppercase.
7. Write a `Function` that returns the length of a string.
8. Create a `Supplier` that supplies a random number between 1 and 100

---

1. Use `Predicate` to filter names starting with "A" from a list.
2. Write a lambda to calculate the square of a number using `UnaryOperator`.
3. Use `BiFunction` to concatenate first name and last name.
4. Implement a `BiPredicate` that checks if a string is longer than a given length.
5. Use `Stream` and `Lambda` to print all even numbers from a list.

---

1. Create a `BinaryOperator` that returns the smaller of two integers.
2. Use `BiConsumer` to print a student's name with their score.
3. Filter a list of integers and print only prime numbers using `Predicate` and `Stream`.
4. Use a `Map<String, Integer>` of student names and marks. Use lambda to print students scoring more than 75.
5. Write a lambda that returns the factorial of a number using a functional interface.

**Examples of using `Comparator` with Lambda Expressions** in Java 8:

---

## 1. Sort Strings Alphabetically

```
List<String> names = Arrays.asList("Zara", "Ankit", "Geet");
names.sort((s1, s2) → s1.compareTo(s2));
```

```
System.out.println(names); // [Ankit, Geet, Zara]
```

## 2. Sort Strings by Length

```
List<String> names = Arrays.asList("Geet", "Anjali", "Abhi");
names.sort((s1, s2) → Integer.compare(s1.length(), s2.length()));
System.out.println(names); // [Geet, Abhi, Anjali]
```

## 3. Sort Custom Objects by Age

```
class Employee {
    String name;
    int age;
    Employee(String name, int age) { this.name = name; this.age = age; }
}

List<Employee> list = Arrays.asList(
    new Employee("Geet", 28),
    new Employee("Raj", 24),
    new Employee("Amit", 30)
);

list.sort((e1, e2) → Integer.compare(e1.age, e2.age));

for (Employee e : list)
    System.out.println(e.name + " - " + e.age);
```

## 4. Sort in Reverse Order (Descending)

```
List<Integer> numbers = Arrays.asList(5, 1, 8, 3);
numbers.sort((a, b) -> b - a);
System.out.println(numbers); // [8, 5, 3, 1]
```

### 5. Sort by Multiple Conditions (e.g., by Name, then Age)

```
list.sort((e1, e2) -> {
    int nameCompare = e1.name.compareTo(e2.name);
    return nameCompare != 0 ? nameCompare : Integer.compare(e1.age, e2.ag
e);
});
```

# Default & Static Methods in Interfaces

## 1. Why Introduced?

Before Java 8, **interfaces could not have method bodies**. Java 8 introduced:

- `default` methods → allow interfaces to have **concrete** methods (backward compatibility).
- `static` methods → allow interfaces to define **utility/helper methods**.

## 2. `default` Method

### Syntax:

```
default return_type methodName() {
    // implementation
}
```

### Purpose:

- Provide a **default implementation** to all classes implementing the interface.
- Avoid **breaking existing code** when new methods are added to interfaces.

### Example:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle is starting...");
    }
}

class Car implements Vehicle {}

public class Test {
    public static void main(String[] args) {
        new Car().start();  // Output: Vehicle is starting...
    }
}
```

---

## 3. `static` Method

### 🔷 Syntax:

```
static return_type methodName() {
    // implementation
}
```

### Purpose:

- Acts like **utility/helper methods** (like in `Collections` or `Math`).
- Can be called using the **interface name only**.

### Example:

```
interface Calculator {
    static int add(int a, int b) {
        return a + b;
    }
}

public class Test {
    public static void main(String[] args) {
        System.out.println(Calculator.add(5, 3));  // Output: 8
    }
}
```

---

## 4. Use Case of Default Method: Multiple Inheritance Problem

```
interface A {
    default void show() {
        System.out.println("From A");
    }
}

interface B {
    default void show() {
        System.out.println("From B");
    }
}
```

```
class Test implements A, B {
    public void show() {
        A.super.show(); // or B.super.show()
    }

    public static void main(String[] args) {
        new Test().show(); // Output: From A
    }
}
```

You **must override** `show()` to resolve ambiguity.

---

### 5. Key Points

| Feature | `default` method | `static` method |
|---|---|---|
| Inherited? | Yes (to implementing class) | No |
| Can override? | Yes | No |
| Call from object? | Yes | No |
| Call from interface? | No (not directly) | Yes |

---

## Tricky Questions (For Practice & Interviews)

1. Can you call a default method using the interface name?
2. What happens if two interfaces have same default method?
3. Can we override a static method of an interface?
4. Can an interface have both static and default methods?
5. Can abstract classes have default methods?
6. What will happen if a class does **not override** a conflicting default method?

---

## Practice Examples

## Example 1: Interface with both methods

```
interface Printer {
    default void print() {
        System.out.println("Printing document...");
    }

    static void welcome() {
        System.out.println("Welcome to Printer Interface");
    }
}

class MyPrinter implements Printer {}

public class Demo {
    public static void main(String[] args) {
        Printer.welcome();            // static method
        new MyPrinter().print();      // default method
    }
}
```

## Example 2: Conflicting default methods

```
interface A { default void show() { System.out.println("A"); } }
interface B { default void show() { System.out.println("B"); } }

class C implements A, B {
    public void show() {
        A.super.show();  // or B.super.show()
    }
}
```

## Why Were `default` and `static` Methods Introduced in Interfaces in Java 8?

### 1. Backward Compatibility

- Before Java 8, adding a new method in an interface would **break all implementing classes**.
- `default` methods allow you to **add new functionality** to interfaces **without affecting existing code**.

*Example:*

```
interface OldInterface {
    void existingMethod();
    // void newMethod(); // ❌ Adding this breaks all implementing classes
}

→ Instead:

default void newMethod() {
    // safe default implementation
}
```

### 2. Support for Functional Programming

- Java 8 introduced **Lambda Expressions** and **Stream API**.
- These APIs rely heavily on functional interfaces, which needed **flexible interfaces** with built-in logic.

### 3. Reduce Utility Classes

- Instead of using utility classes like `Collections`, `Math`, or `Arrays`, now interfaces can **directly provide static utility methods**.

*Example:*

```
interface MathUtil {
    static int square(int x) {
        return x * x;
    }
}
```

---

### 4. Multiple Inheritance of Behavior

- Enables **code reuse** by allowing interfaces to contain behavior (via `default` methods).
- Brings interfaces closer to **traits/mixins** seen in other languages (like Scala, Kotlin).

---

### Summary:

| Problem Before Java 8 | Solution with Java 8 |
|---|---|
| Can't add new methods in interface | `default` methods |
| Interfaces had no method body | `default` + `static` methods |
| Required external utility classes | `static` methods in interfaces |
| No behavioral inheritance in interfaces | Multiple `default` methods with resolution |

## Java 8 – `java.time` Package (Date & Time API)1. Why `java.time` was introduced?

---

The **old date/time classes** (`java.util.Date`, `java.util.Calendar`) had problems:

- Mutable objects (not thread-safe)
- Confusing APIs (month starts from 0)
- No support for time zones or formatting out-of-the-box

Java 8 introduced `java.time` package for a **modern, clean, immutable, and thread-safe API**.

## 2. Core Classes in `java.time`

| Class | Description |
|---|---|
| `LocalDate` | Date only (YYYY-MM-DD) |
| `LocalTime` | Time only (HH:MM:SS) |
| `LocalDateTime` | Date + Time |
| `ZonedDateTime` | Date + Time + Time Zone |
| `Period` | Difference in dates (days, months) |
| `Duration` | Difference in time (hours, seconds) |
| `DateTimeFormatter` | For formatting and parsing |

## 3. `LocalDate` Example

```
import java.time.LocalDate;

public class Demo {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        LocalDate birthday = LocalDate.of(1995, 5, 20);

        System.out.println("Today: " + today);
        System.out.println("Birthday: " + birthday);
        System.out.println("Day of week: " + birthday.getDayOfWeek());
    }
}
```

## 4. `LocalTime` Example

```
import java.time.LocalTime;

public class TimeExample {
    public static void main(String[] args) {
        LocalTime now = LocalTime.now();
        System.out.println("Current Time: " + now);
    }
}
```

## 5. `LocalDateTime` Example

```
import java.time.LocalDateTime;

public class DateTimeExample {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.now();
        System.out.println("Date & Time: " + dt);
    }
}
```

## 6. `ZonedDateTime` Example

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class ZoneExample {
    public static void main(String[] args) {
        ZonedDateTime zoned = ZonedDateTime.now(ZoneId.of("Asia/Kolkata"));
```

```
        System.out.println("Zoned Date & Time: " + zoned);
    }
}
```

## 7. `Period` Example (Date Difference)

```
import java.time.LocalDate;
import java.time.Period;

public class PeriodExample {
    public static void main(String[] args) {
        LocalDate past = LocalDate.of(2020, 1, 1);
        LocalDate now = LocalDate.now();
        Period diff = Period.between(past, now);
        System.out.println("Years: " + diff.getYears());
        System.out.println("Months: " + diff.getMonths());
        System.out.println("Days: " + diff.getDays());
    }
}
```

## 8. `Duration` Example (Time Difference)

```
import java.time.Duration;
import java.time.LocalTime;

public class DurationExample {
    public static void main(String[] args) {
        LocalTime start = LocalTime.of(9, 30);
        LocalTime end = LocalTime.now();
        Duration d = Duration.between(start, end);
```

```
        System.out.println("Duration in minutes: " + d.toMinutes());
    }
}
```

## 9. `DateTimeFormatter` Example (Formatting)

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class FormatExample {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.now();
        DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy
HH:mm:ss");
        System.out.println("Formatted: " + dt.format(format));
    }
}
```

## 10. Advantages of `java.time` API

| Feature | Benefit |
|---|---|
| Immutable classes | Thread-safe |
| Clear class naming | Easy to understand |
| Zone/time support | `ZonedDateTime`, `ZoneId` |
| Period/Duration | Easily calculate differences |
| Better formatting | `DateTimeFormatter` over `SimpleDateFormat` |
| Fluent API | Method chaining is intuitive |

# Java 8 – `Optional<T>` Class

## 1. What is `Optional`?

`Optional` is a **container object** that may or may not contain a **non-null value**.

It was introduced to **avoid** `NullPointerException` and enforce **null safety** in Java.

---

## ✅ 2. Why was `Optional` introduced?

| Problem Before Java 8 | How `Optional` Helps |
| --- | --- |
| `NullPointerException` everywhere | Avoids nulls by design |
| Tedious null checks | Cleaner, functional code |
| Poor readability | Provides expressive intent (value may be absent) |

---

## 3. Declaration & Creation

### Using `of()`

```
Optional<String> opt = Optional.of("Geet");
```

> ❗ Throws NullPointerException if value is null.

### Using `ofNullable()`

```
Optional<String> opt = Optional.ofNullable(null); // safe
```

### Using `empty()`

```
Optional<String> opt = Optional.empty();
```

---

## 4. Common Methods in `Optional<T>`

| Method | Description |
|---|---|
| `isPresent()` | Returns `true` if value is present |
| `ifPresent(Consumer)` | Executes lambda if value exists |
| `get()` | Returns value; throws `NoSuchElementException` if empty |
| `orElse(T)` | Returns value if present, else default |
| `orElseGet(Supplier)` | Same as `orElse` but lazily evaluates |
| `orElseThrow()` | Throws exception if value not present |
| `map(Function)` | Transforms the value |
| `flatMap(Function)` | Like map but returns Optional |
| `filter(Predicate)` | Returns Optional if condition matches |

## 5. Examples

### isPresent() and get()

```
Optional<String> opt = Optional.of("Java");
if (opt.isPresent()) {
    System.out.println(opt.get()); // Java
}
```

### orElse()

```
String value = Optional.ofNullable(null).orElse("Default");
System.out.println(value); // Default
```

### orElseGet()

```
String value = Optional.ofNullable(null)
    .orElseGet(() → "Generated Default");
System.out.println(value); // Generated Default
```

## orElseThrow()

```
String name = Optional.ofNullable(null)
    .orElseThrow(() → new RuntimeException("Name not found"));
```

## ifPresent()

```
Optional<String> opt = Optional.of("GeetCode");
opt.ifPresent(val → System.out.println(val.length())); // 8
```

## map() and flatMap()

```
Optional<String> opt = Optional.of("Geet");
int len = opt.map(String::length).orElse(0);
System.out.println(len); // 4
```

## filter()

```
Optional<String> opt = Optional.of("Java");
opt = opt.filter(val → val.startsWith("J"));
```

```
System.out.println(opt.isPresent()); // true
```

## 6. Real-Time Use Case

In Service Layer:

```
public Optional<Employee> findById(int id) {
    return employeeRepo.findById(id); // may return empty
}

public String getEmployeeName(int id) {
    return findById(id)
            .map(Employee::getName)
            .orElse("Employee Not Found");
}
```

## 7. When NOT to Use Optional

- **Don't use `Optional` as method parameters.**
- **Don’t use `Optional` in entity classes or fields (especially in JPA).**
- **Avoid wrapping collections in `Optional`** — just return an empty collection.

## 8. Interview-Tricky Questions

1. What's the difference between `orElse()` and `orElseGet()`?
2. Can `Optional.get()` throw exceptions? When?
3. When would you prefer `Optional` over traditional null checks?
4. Can `Optional` replace every `null`? Why or why not?
5. How is `flatMap()` different from `map()` in Optional?
6. Why is `Optional` not Serializable?

## 9. Summary Table

| Use Case | Method to Use |
| --- | --- |
| Check if value exists | `isPresent()` |
| Execute code if value exists | `ifPresent()` |
| Return default if null | `orElse()` / `orElseGet()` |
| Throw exception if empty | `orElseThrow()` |
| Transform value | `map()` |
| Transform + flatten Optionals | `flatMap()` |
| Conditional filtering | `filter()` |

# Ways to Handle `NullPointerException` in Java

## 1. Explicit Null Checks

```
if (obj != null) {
    obj.doSomething();
}
```

- **Simple & direct**, but becomes messy with deep nesting.
- Useful for small code blocks.
- Not scalable in large systems.

## 2. `try-catch` Block

```
try {
    obj.doSomething();
} catch (NullPointerException e) {
    System.out.println("Handled NPE");
```

```
}
```

- Catches NPE but **not recommended** for logic control.
- Bad practice to handle nulls via exception.

---

## 3. Use `Optional<T>` (Java 8+)

```
Optional.ofNullable(obj).ifPresent(o → o.doSomething());
```

- **Recommended** modern way to handle potential nulls.
- Improves readability, avoids runtime crash.
- Slight learning curve for beginners.

---

## 4. Use `Objects.requireNonNull()`

```
this.name = Objects.requireNonNull(name, "Name can't be null");
```

- Defensive coding: fails fast during development.
- Useful for constructor/method parameter validation.

---

## 5. Default Values / Ternary Operator

```
String result = str != null ? str : "Default";
```

- Useful for assigning defaults.

- Simple and readable for small cases.

### 6. Apache Commons or Guava Utils

```
String safe = StringUtils.defaultIfEmpty(input, "default");
```

- Helps avoid boilerplate.
- Adds external dependency.

## Recommended Approach

| Situation | Best Practice |
|---|---|
| General object usage | Use `Optional<T>` |
| Constructor/method parameters | Use `Objects.requireNonNull()` |
| Simple defaults | Ternary or `Optional.orElse()` |
| Utility-based projects | Consider Apache `StringUtils` |
| Avoiding messy null checks | Avoid `try-catch` for NPEs |

### Final Answer:

> The most recommended modern approach is to use Optional<T> for return values and Objects.requireNonNull() for parameter validation. Avoid using try-catch to handle nulls.

## Java 8 – Method References ( `::` Operator)

### 1. What is a Method Reference?

A **method reference** is a **shorter, cleaner** way to refer to a method **without executing it**.

It's used as a **shortcut for lambda expressions** that only call an existing method.

> It improves code readability and is often used with functional interfaces (e.g., Predicate, Function, etc.).

## 2. Syntax:

```
ClassName::methodName
```

## 3. Why use Method References?

| Lambda Expression | Equivalent Method Reference |
| --- | --- |
| `(s) → s.toUpperCase()` | `String::toUpperCase` |
| `(a, b) → Math.max(a, b)` | `Math::max` |
| `() → new Student()` | `Student::new` |

- **Less code, more clarity**

## 4. Types of Method References

| Type | Syntax Example | Description |
| --- | --- | --- |
| **1. Static Method** | `ClassName::staticMethod` | Refers to a static method |
| **2. Instance Method (Object)** | `object::instanceMethod` | Refers to a method on a specific object |
| **3. Instance Method (Class)** | `ClassName::instanceMethod` | First parameter becomes the caller |
| **4. Constructor Reference** | `ClassName::new` | Refers to a constructor |

## 5. Examples 1. Static Method Reference

```
class Utility {
    public static void greet() {
        System.out.println("Hello!");
    }
}

Runnable r = Utility::greet;
r.run();  // Output: Hello!
```

## 2. Instance Method Reference (on Object)

```
class Printer {
    public void print(String msg) {
        System.out.println(msg);
    }
}

Printer printer = new Printer();
Consumer<String> c = printer::print;
c.accept("Welcome!");  // Output: Welcome!
```

## 3. Instance Method Reference (on Class)

```
List<String> names = Arrays.asList("Geet", "Ankit", "Zara");

// Using lambda:
// names.forEach(name → System.out.println(name));
```

```
// Using method reference:
names.forEach(System.out::println);
```

## 4. Constructor Reference

```
interface StudentCreator {
    Student create();
}

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

StudentCreator sc = Student::new;
sc.create(); // Output: Student object created
```

## 6. Real-Time Use Case

```
List<String> list = Arrays.asList("apple", "banana", "cherry");

list.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

## 7. Interview FAQs

1. What is the difference between lambda and method reference?
2. When should you use method reference over lambda?
3. Can you use method reference for overloaded methods?
4. Can method references work with arguments?
5. What is the difference between `object::method` and `ClassName::method`?

---

## Summary

| Lambda | Method Reference |
|---|---|
| `str -> str.toUpperCase()` | `String::toUpperCase` |
| `(a, b) -> Math.max(a, b)` | `Math::max` |
| `() -> new MyObject()` | `MyObject::new` |
| `x -> x.print()` | `MyObject::print` |

**Method reference examples** categorized by type, so your students can easily grasp each use case:

---

## 1. Static Method Reference

```
import java.util.function.BiFunction;

public class StaticExample {
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> sum = StaticExample::add;
        System.out.println(sum.apply(5, 3)); // Output: 8
    }
}
```

## 2. Instance Method Reference (specific object)

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

class Printer {
    public void print(String msg) {
        System.out.println(msg);
    }
}

public class InstanceExample {
    public static void main(String[] args) {
        Printer printer = new Printer();
        Consumer<String> messagePrinter = printer::print;
        messagePrinter.accept("Hello Method Reference!"); // Output: Hello Meth
od Reference!
    }
}
```

## 3. Instance Method Reference (class name)

```
import java.util.Arrays;
import java.util.List;

public class ClassInstanceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Geet", "Ankit", "Zara");

        // Lambda: names.forEach(name → System.out.println(name));
        names.forEach(System.out::println); // Output: Geet Ankit Zara
    }
```

```
}
```

## 4. Constructor Reference

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

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

public class ConstructorExample {
    public static void main(String[] args) {
        Supplier<Student> studentSupplier = Student::new;
        studentSupplier.get(); // Output: Student created!
    }
}
```

## 5. Custom Functional Interface with Constructor

```
interface EmployeeFactory {
    Employee create(String name, int age);
}

class Employee {
    String name;
    int age;

    Employee(String name, int age) {
```

```
        System.out.println("Employee: " + name + ", Age: " + age);
    }
}

public class CustomConstructorRef {
    public static void main(String[] args) {
        EmployeeFactory factory = Employee::new;
        factory.create("Geetanjali", 28); // Output: Employee: Geetanjali, Age: 28
    }
}
```

## Nashorn JavaScript Engine – Java 8

### What is Nashorn?

**Nashorn** is a **JavaScript engine** introduced in **Java 8** that allows you to **run JavaScript code inside Java applications**.

> It replaced the older Rhino engine and offers better performance and ECMAScript compliance.

### Purpose of Nashorn:

- Run JavaScript code from within Java (like embedded scripting).
- Enable interaction between Java and JavaScript code.
- Used in server-side templating, data transformation, and scripting scenarios.

### Example: Executing JavaScript in Java

```
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;
```

```
public class NashornDemo {
    public static void main(String[] args) throws ScriptException {
        ScriptEngine engine = new ScriptEngineManager().getEngineByName("n
ashorn");
        engine.eval("print('Hello from JavaScript!');");
    }
}
```

> Output: Hello from JavaScript!**Using Java Objects in JavaScript**

---

```
engine.eval("var list = new java.util.ArrayList(); list.add('One'); print(list.get
(0));");
```

---

### Deprecation Note:

- Nashorn was **deprecated in Java 11** and **removed in Java 15** due to lack of use and better alternatives like **GraalVM**.

---

### Use Cases (When it was used):

- Scripting logic in business rules
- Template rendering engines
- Small embedded calculations in large Java apps

## Java 8 – Stream API

---

### 1. What is Stream API?

The **Stream API** allows processing of **collections and arrays** in a **functional, declarative**, and **parallelizable** way.

> It supports filtering, mapping, reducing, and collecting operations on data.

## 2. Why Use Stream API?

| Traditional Java | Stream API |
| --- | --- |
| Verbose (loops) | Concise (chained operations) |
| External Iteration | Internal Iteration |
| Not Parallel Friendly | Easily Parallelizable |
| Imperative | Declarative (what not how) |

## 3. Stream vs Collection

| Feature | Collection | Stream |
| --- | --- | --- |
| Stores Data | Yes | No (pipeline of data) |
| Consumes Once | No | Yes (one-time use) |
| External Iteration | Yes (loops) | No |
| Lazy Evaluation | No | Yes |

## 4. Stream Creation

```
List<Integer> list = Arrays.asList(1, 2, 3);
Stream<Integer> stream = list.stream();

Stream<String> s = Stream.of("A", "B", "C");
```

## 5. Key Stream Methods

| Method | Description |
| --- | --- |
| `filter()` | Filters elements based on condition |
| `map()` | Transforms each element |
| `forEach()` | Performs action on each element |
| `collect()` | Collects results into list/set/map |
| `count()` | Returns count of elements |
| `sorted()` | Sorts elements |
| `limit(n)` | Returns first n elements |
| `skip(n)` | Skips first n elements |
| `reduce()` | Aggregates to a single result |
| `distinct()` | Removes duplicates |

## 6. Examples

### Filter even numbers

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

### Map to square

```
List<Integer> square = list.stream().map(n -> n * n).collect(Collectors.toList());
```

### Count strings starting with 'G'

```
long count = names.stream().filter(n -> n.startsWith("G")).count();
```

### Sort list

```
List<String> sorted = names.stream().sorted().collect(Collectors.toList());
```

### Reduce to sum

```
int sum = list.stream().reduce(0, (a, b) → a + b);
```

---

## 7. Terminal vs Intermediate Operations

| Intermediate | Terminal |
|---|---|
| `filter()` | `collect()` |
| `map()` | `forEach()` |
| `sorted()` | `count()` |
| `limit()` | `reduce()` |

---

# Tricky Interview Questions

1. What is the difference between `map()` and `flatMap()`?
2. Can a stream be reused? If not, why?
3. What's the difference between `Stream.of()` and `Arrays.stream()`?
4. When should you prefer `parallelStream()`?
5. What does lazy evaluation mean in the context of streams?

---

# Coding Questions for Practice

1. Write a program to square each number in a list using `map()`.
2. Count the number of strings with length > 5 in a list.
3. Filter a list of names starting with "A".

1. Sort a list of `Employee` by salary using streams.
2. Convert a list of strings to uppercase and collect them into a Set.
3. Given a list of words, return the longest one using `reduce()`.

1. From a list of `Order` objects, filter out orders that are `delivered` and cost > ₹1000.
2. Group a list of `Employee` by department using `Collectors.groupingBy()`.
3. Use `flatMap()` to flatten a list of lists.
4. Combine `filter()`, `map()`, and `reduce()` to calculate the sum of squares of even numbers.

### Bonus: Real-World Case

```
Map<String, Long> nameCount = names.stream()
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

## Advanced Stream API Examples with Answers

### 1. Find Second Highest Salary

```
List<Integer> salaries = Arrays.asList(30000, 50000, 70000, 50000, 90000);

Optional<Integer> secondHighest = salaries.stream()
    .distinct()
    .sorted(Comparator.reverseOrder())
```

```
    .skip(1)
    .findFirst();

System.out.println("Second Highest Salary: " + secondHighest.orElse(-1));
```

## 2. Group Employees by Department

```
Map<String, List<Employee>> byDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));

byDept.forEach((dept, list) -> {
    System.out.println(dept + " : " + list.size());
});
```

## 3. Department-wise Total Salary

```
Map<String, Integer> deptSalary = employees.stream()
    .collect(Collectors.groupingBy(
        Employee::getDepartment,
        Collectors.summingInt(Employee::getSalary)
    ));
```

## 4. Employee with Max Salary

```
Optional<Employee> maxSalaryEmp = employees.stream()
    .max(Comparator.comparingInt(Employee::getSalary));
```

```
maxSalaryEmp.ifPresent(System.out::println);
```

## 5. Average Salary per Department

```
Map<String, Double> avgSalary = employees.stream()
    .collect(Collectors.groupingBy(
        Employee::getDepartment,
        Collectors.averagingInt(Employee::getSalary)
    ));
```

## 6. List All Employee Names in a Comma-Separated String

```
String names = employees.stream()
    .map(Employee::getName)
    .collect(Collectors.joining(", "));

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

## 7. Filter Employees Who Joined After 2020

```
List<Employee> recent = employees.stream()
    .filter(e -> e.getJoinYear() > 2020)
    .collect(Collectors.toList());
```

## 8. Group by Department and Find Highest Salary Employee in Each

```
Map<String, Optional<Employee>> topEarners = employees.stream()
    .collect(Collectors.groupingBy(
        Employee::getDepartment,
        Collectors.maxBy(Comparator.comparingInt(Employee::getSalary))
    ));
```

## 9. Sort Employees by Department and then Salary Descending

```
List<Employee> sorted = employees.stream()
    .sorted(Comparator.comparing(Employee::getDepartment)
        .thenComparing(Employee::getSalary, Comparator.reverseOrder()))
    .collect(Collectors.toList());
```

## 10. FlatMap Example – Flatten List of Skills

```
List<List<String>> skillSets = Arrays.asList(
    Arrays.asList("Java", "Spring"),
    Arrays.asList("SQL", "MongoDB"),
    Arrays.asList("React", "Angular")
);

List<String> allSkills = skillSets.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
```

```
System.out.println(allSkills); // [Java, Spring, SQL, MongoDB, React, Angular]
```

# Java 8 – `Collectors` API (Used with Stream API)

## 1. What is `Collectors`?

`Collectors` is a **utility class** in `java.util.stream.Collectors` that provides **predefined methods** to collect the result of stream operations into various data structures.

> It works with terminal operation collect() in the Stream API. 2. Why Use `Collectors`?

| Without Collectors | With Collectors |
| --- | --- |
| Manual result processing | Predefined collection logic |
| Verbose and complex | Concise and readable |
| No grouping/partitioning | Built-in grouping, joining |

## 3. Syntax:

```
stream.collect(Collectors.methodName());
```

## 4. Commonly Used `Collectors` Methods

| Method | Description |
| --- | --- |
| `toList()` | Collect elements into a `List` |
| `toSet()` | Collect elements into a `Set` |
| `toMap(k -> ..., v -> ...)` | Collect into a `Map` |
| `joining(delimiter)` | Concatenate strings |

| | |
|---|---|
| `counting()` | Count elements |
| `summingInt()` | Sum integer properties |
| `averagingInt()` | Average of integer values |
| `groupingBy()` | Group by a classifier function |
| `partitioningBy()` | Partition into two groups based on condition |
| `mapping()` | Additional mapping after grouping |
| `maxBy()` / `minBy()` | Find max/min with comparator |
| `reducing()` | Generalized reduction |

## 5. Examples of `Collectors` in Action

### Example 1: Collect to List

```
List<String> names = Stream.of("Geet", "Ankit", "Raj")
    .collect(Collectors.toList());
```

### Example 2: Collect to Set

```
Set<String> uniqueNames = Stream.of("A", "B", "A")
    .collect(Collectors.toSet());
```

### Example 3: Count Elements

```
long count = Stream.of("Java", "Python", "Java")
    .collect(Collectors.counting());
```

## Example 4: Join Strings

```
String joined = Stream.of("Java", "8", "Stream")
    .collect(Collectors.joining(" "));
System.out.println(joined); // Java 8 Stream
```

## Example 5: Sum and Average

```
int sum = employees.stream()
    .collect(Collectors.summingInt(Employee::getSalary));

double avg = employees.stream()
    .collect(Collectors.averagingInt(Employee::getSalary));
```

## Example 6: Group by Department

```
Map<String, List<Employee>> byDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));
```

## Example 7: Partition by Salary > 50000

```
Map<Boolean, List<Employee>> partitioned = employees.stream()
    .collect(Collectors.partitioningBy(e → e.getSalary() > 50000));
```

### Example 8: Collect to Map

```
Map<Integer, String> empMap = employees.stream()
    .collect(Collectors.toMap(Employee::getId, Employee::getName));
```

### Example 9: Max Salary by Department

```
Map<String, Optional<Employee>> topEarners = employees.stream()
    .collect(Collectors.groupingBy(
        Employee::getDepartment,
        Collectors.maxBy(Comparator.comparingInt(Employee::getSalary))
    ));
```

### Example 10: Reduce Using `reducing()`

```
int totalSalary = employees.stream()
    .collect(Collectors.reducing(0, Employee::getSalary, Integer::sum));
```

## Tricky Interview Questions

1. What's the difference between `groupingBy()` and `partitioningBy()`?
2. Can `toMap()` fail? (Yes, if duplicate keys — throws `IllegalStateException`)
3. How does `reducing()` differ from `reduce()`?
4. What is `mapping()` used for inside `groupingBy()`?
5. What is the default collector returned by `Collectors.groupingBy()`?

## Summary

| Use Case | Use This Collector |
|---|---|
| To get a `List` | `Collectors.toList()` |
| To get a `Set` | `Collectors.toSet()` |
| Combine strings | `Collectors.joining()` |
| Grouping by a property | `Collectors.groupingBy()` |
| Summing/averaging values | `summingInt()`, `averagingInt()` |
| Partition true/false logic | `Collectors.partitioningBy()` |

# Task

Top 50 Java 8 Coding Questions for Interviews (with Java 8 Features)

Lambda Expressions

1. Create a lambda to add two numbers.
2. Use lambda to sort a list of strings alphabetically.
3. Write a program using lambda to count strings starting with a given letter.
4. Replace anonymous inner class with lambda for Runnable.
5. Write a lambda expression to reverse a string.

Functional Interfaces

6. Create your own functional interface and implement using lambda.
7. Use Predicate to check if a number is even.
8. Use Function to convert string to its length.
9. Use BiFunction to concatenate two strings.
10. Use Consumer to print strings in uppercase.

Method References

11. Use static method reference to print a message.
12. Use instance method reference from an object.
13. Use class method reference to print a list.
14. Use constructor reference to create object.
15. Use method reference inside stream forEach.

Stream API

16. Find even numbers from a list.
17. Square and filter numbers greater than 10.
18. Count the number of empty strings.
19. Find the longest string in a list.
20. Join a list of strings with commas.
21. Convert list to set using streams.
22. Get list of employees with salary > 50000.
23. Group employees by department.
24. Find second highest salary using streams.
25. Get average salary by department.
26. Sort list of employees by name.
27. Find max and min salary using streams.
28. Partition list of numbers into even and odd.
29. Flatten a list of lists using flatMap.
30. Remove duplicate elements from a list.

Optional

31. Create an Optional object and print value.
32. Use Optional.ofNullable() to avoid NPE.
33. Use ifPresent() to print value.
34. Use orElse and orElseGet.
35. Use Optional with map and filter.

Default & Static Methods in Interface

36. Create interface with default and static methods.
37. Override default method in implementing class.
38. Call static method from interface.
39. Resolve conflict between two interfaces with same default method.

Date & Time API

40. Print current date using LocalDate.
41. Calculate age using Period.
42. Get current time using LocalTime.
43. Format date using DateTimeFormatter.
44. Parse date string into LocalDate.

Collectors

45. Use Collectors.groupingBy() and Collectors.summingInt().