**Lambda Expression**

**What is lambda operator(->) in Java?**

The -> is called the **lambda operator** or **arrow operator**.

It separates the **parameters** on the left from the **body** of the lambda expression on the right.

**Lambda Expression Syntax:**

(parameters) -> expression\_or\_block

**Examples**

**1. Simple Expression**

(x) -> x \* x

This is a function that takes one argument x and returns x \* x.

**2. Used with Stream API**

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

names.stream()

.filter(name -> name.startsWith("A"))

.forEach(name -> System.out.println(name));

Explanation:

* name -> name.startsWith("A") → Predicate (returns true if name starts with A)
* name -> System.out.println(name) → Consumer (prints each name)

**3. With Multiple Parameters**

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

Takes two integers, returns their sum.

**4. Block Body with Return Statement**

(x, y) -> {

int sum = x + y;

return sum;

}

You can use curly braces {} and a return keyword for multi-line logic.

**Where is -> used?**

* With **functional interfaces** (interfaces with a single abstract method)
* Commonly used with:
  + Stream API (map, filter, reduce, forEach, etc.)
  + Runnable, Comparator, Callable, Predicate, Consumer, Function interfaces

**Example with Comparator**

List<String> list = Arrays.asList("Banana", "Apple", "Mango");

list.sort((a, b) -> a.compareTo(b)); // Lambda replaces Comparator

**Summary**

| **Symbol** | **Meaning** | **Used in** |
| --- | --- | --- |
| -> | Lambda / arrow operator | Lambda expressions |
| Left | Parameters | Like x, or (x, y) |
| Right | Expression or logic block | Function body |

Below is a **comprehensive collection of Java lambda expression examples** covering all common use cases, grouped by functionality.

## 1. Basic Lambda Expression

Runnable r = () -> System.out.println("Hello from Lambda!");

r.run();

Let's break down the following code step by step:

Runnable r = () -> System.out.println("Hello from Lambda!");

r.run();

**1. Runnable Interface**

Runnable is a **functional interface** in Java, defined as:

@FunctionalInterface

public interface Runnable {

void run();

}

* It has **only one abstract method**: void run().
* Functional interfaces can be used with lambda expressions.
* Normally used to define tasks to run in a **thread**.

**2. Lambda Expression**

**() -> System.out.println("Hello from Lambda!");**

This is a lambda expression that:

* Takes **no arguments** (empty ()).
* Executes the code System.out.println("Hello from Lambda!") when invoked.

It matches the signature of the Runnable interface's run() method (no parameters, no return value).

**3. Assigning the Lambda to a Runnable**

Runnable r = () -> System.out.println("Hello from Lambda!");

* You're assigning the lambda to a variable r of type Runnable.
* Internally, Java treats this as an **anonymous class implementation** of Runnable.

**4. Executing the Lambda**

r.run();

* This calls the run() method of the Runnable instance.
* Since the lambda is the implementation of run(), it prints:

Hello from Lambda!

**Equivalent without Lambda**

This code:

Runnable r = () -> System.out.println("Hello from Lambda!");

Is equivalent to:

Runnable r = new Runnable() {

@Override

public void run() {

System.out.println("Hello from Lambda!");

}

};

**Summary**

| **Part** | **Meaning** |
| --- | --- |
| Runnable | Functional interface with void run() |
| () -> ... | Lambda with no parameters |
| r.run() | Executes the lambda |
| Output | "Hello from Lambda!" printed to console |

## 2. Lambda with Parameters

interface Adder {

int add(int a, int b);

}

public class Test {

public static void main(String[] args) {

Adder adder = (a, b) -> a + b;

System.out.println(adder.add(5, 3)); // Output: 8

}

}

### Full Code:

interface Adder {

int add(int a, int b); // Functional Interface

}

public class Lambda0003 {

public static void main(String[] args) {

Adder adder = (a, b) -> a + b; // Lambda expression

System.out.println(adder.add(20, 10)); // Output: 30

}

}

### Explanation:

#### 1**. interface Adder**

interface Adder {

int add(int a, int b);

}

* This is a **functional interface**, because it has only **one abstract method**.
* By Java 8 rules, it qualifies to be implemented using a lambda expression.

You can optionally add @FunctionalInterface above it for clarity and compiler support.

#### 2**. Lambda Expression**

Adder adder = (a, b) -> a + b;

* This is the **lambda** that provides the implementation of the add() method.
* (a, b) are two parameters, and a + b is the return value.
* It **automatically matches** the add(int, int) method of the Adder interface.

#### 3**. Invocation**

System.out.println(adder.add(20, 10)); // Prints 30

* This calls the add() method with arguments 20 and 10.
* It returns 30, which is printed.

### Output:

30

### Equivalent Code Without Lambda:

Adder adder = new Adder() {

@Override

public int add(int a, int b) {

return a + b;

}

};

System.out.println(adder.add(20, 10));

As you can see, using a lambda makes your code **shorter** and **easier to read**.

## 3. Using Functional Interfaces from java.util.function

### a) Function<T, R> – One input, one output

**import** java.util.function.Function;

**public** **class** Lambda004 {

**public** **static** **void** main(String[] args) {

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

System.***out***.println(square.apply(6)); // Output: 36

}

}

### Explanation:

#### **1. Function<Integer, Integer>**

* This is a **functional interface** from java.util.function.
* It represents a function that:
  + Takes **one input** of type Integer.
  + Returns **one output** of type Integer.

#### **2. Lambda Expression:**

x -> x \* x

* This lambda takes an integer x and returns its square.

#### **3. .apply(6)**

* This method executes the function with input 6.
* It computes 6 \* 6 = 36 and returns it.

### Output:

36

### Bonus Tip:

You could even chain functions like this:

Function<Integer, Integer> doubleIt = x -> x \* 2;

Function<Integer, Integer> squareThenDouble = square.andThen(doubleIt);

System.out.println(squareThenDouble.apply(5)); // (5\*5)=25, then 25\*2=50

### b) BiFunction<T, U, R> – Two inputs, one output

**import** java.util.function.BiFunction;

**public** **class** Lambda005 {

**public** **static** **void** main(String[] args) {

BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a \* b;

System.***out***.println(multiply.apply(4, 5)); // Output: 20

}

}

### Explanation:

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

This is a **built-in functional interface** from java.util.function.  
It represents a function that:

* Takes **two input arguments**: T and U
* Returns a result of type R

In your case:

BiFunction<Integer, Integer, Integer>

* Inputs: Integer, Integer
* Output: Integer

So it models a method like:

Integer apply(Integer a, Integer b);

#### 2. Lambda Expression:

(a, b) -> a \* b

* Takes two integers a and b
* Returns the result of multiplying them

#### 3. Execution:

System.out.println(multiply.apply(4, 5));

* apply(4, 5) → 4 \* 5 → 20

### Output:

20

### Bonus: More BiFunction Examples

#### Add two numbers:

BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;

System.out.println(add.apply(10, 15)); // Output: 25

#### Combine two strings:

BiFunction<String, String, String> concat = (s1, s2) -> s1 + " " + s2;

System.out.println(concat.apply("Hello", "World")); // Output: Hello World

#### Chain BiFunction with andThen():

BiFunction<Integer, Integer, Integer> sum = (x, y) -> x + y;

Function<Integer, String> toString = result -> "Result: " + result;

System.out.println(sum.andThen(toString).apply(7, 8)); //Output: Result: 15

### c) Predicate<T> – Returns boolean

**import** java.util.function.Predicate;

**public** **class** Lambda006 {

**public** **static** **void** main(String[] args) {

Predicate<String> isEmpty = s -> s.isEmpty();

System.***out***.println(isEmpty.test("")); // true

System.***out***.println(isEmpty.test("Hi")); // false

}

}

### What is Predicate<T>?

* It is a **functional interface** from java.util.function.
* It represents a function that takes one argument of type T and **returns a boolean**.

#### Method inside Predicate:

boolean test(T t);

So your lambda:

s -> s.isEmpty()

means:

* Take a string s
* Return true if it’s empty (""), otherwise false

### Output:

true

false

### Bonus: More Predicate Examples

#### 1. Check if a number is even:

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

System.out.println(isEven.test(4)); // true

System.out.println(isEven.test(5)); // false

#### 2. Use negate():

Predicate<String> notEmpty = isEmpty.negate();

System.out.println(notEmpty.test("Hi")); // true

System.out.println(notEmpty.test("")); // false

#### 3. Combine with and() / or():

Predicate<String> hasLength5 = s -> s.length() == 5;

Predicate<String> startsWithH = s -> s.startsWith("H");

Predicate<String> both = hasLength5.and(startsWithH);

System.out.println(both.test("Hello")); // true

System.out.println(both.test("Hi")); // false

### d) BiPredicate<T, U>

import java.util.function.BiPredicate;

BiPredicate<Integer, Integer> isGreater = (a, b) -> a > b;

System.out.println(isGreater.test(5, 3)); // true

### e) Consumer<T> – Consumes input, returns nothing

import java.util.function.Consumer;

Consumer<String> print = s -> System.out.println("Hello " + s);

print.accept("World"); // Hello World

### f) BiConsumer<T, U>

import java.util.function.BiConsumer;

BiConsumer<String, Integer> printAge = (name, age) ->

System.out.println(name + " is " + age + " years old.");

printAge.accept("Alice", 30); // Output: Alice is 30 years old.

### g) Supplier<T> – Supplies output, takes nothing

import java.util.function.Supplier;

Supplier<Double> random = () -> Math.random();

System.out.println(random.get());

### h) UnaryOperator<T> – Function with same input/output type

import java.util.function.UnaryOperator;

UnaryOperator<String> toUpper = s -> s.toUpperCase();

System.out.println(toUpper.apply("java")); // Output: JAVA

### i) BinaryOperator<T> – Two inputs, one output (same type)

import java.util.function.BinaryOperator;

BinaryOperator<Integer> add = (a, b) -> a + b;

System.out.println(add.apply(10, 20)); // Output: 30

## 4. Lambda in Collection Operations

### a) forEach

import java.util.Arrays;

import java.util.List;

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

names.forEach(name -> System.out.println(name));

### b) sort with Comparator

import java.util.\*;

List<String> names = Arrays.asList("Banana", "Apple", "Mango");

names.sort((a, b) -> a.compareToIgnoreCase(b));

System.out.println(names);

### c) removeIf with Predicate

List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

numbers.removeIf(n -> n % 2 == 0);

System.out.println(numbers); // [1, 3, 5]

## 5. Lambda with Threads

new Thread(() -> System.out.println("Running in a thread")).start();

## 6. Lambda with Stream API

import java.util.Arrays;

import java.util.List;

List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);

int sum = nums.stream()

.filter(n -> n % 2 == 0)

.mapToInt(n -> n)

.sum();

System.out.println(sum); // Output: 6 (2+4)

## 7. Lambda for Custom Sorting Objects

import java.util.\*;

class Product {

String name;

double price;

Product(String name, double price) {

this.name = name; this.price = price;

}

public String toString() {

return name + ": " + price;

}

}

public class Test {

public static void main(String[] args) {

List<Product> list = Arrays.asList(

new Product("Laptop", 1200),

new Product("Tablet", 800),

new Product("Phone", 1000)

);

list.sort((p1, p2) -> Double.compare(p1.price, p2.price));

list.forEach(p -> System.out.println(p));

}

}

**Streams**

In Java, **streams** are used to perform **input and output (I/O)** operations.

A **stream** is a sequence of data elements made available over time, and it represents a flow of data from a source to a destination.

Java has two main types of streams:

### 1. ****Byte Streams (****InputStream ****/**** OutputStream****)****

* Used for reading and writing **binary data** (like images, audio, etc.).
* Base classes:
  + InputStream (for reading bytes)
  + OutputStream (for writing bytes)

#### Examples:

**try** {

FileOutputStream fos = **new** FileOutputStream("file.bin");

fos.write(65); // Writes ASCII 'A'

fos.close();

}

**catch**(FileNotFoundException e) {

e.printStackTrace();

}

**catch**(IOException e) {

e.printStackTrace();

}

**try** {

FileInputStream fis = **new** FileInputStream("file.bin");

**int** b = fis.read();

System.***out***.println((**char**)b);

fis.close();

}

**catch**(FileNotFoundException e) {

e.printStackTrace();

}

**catch**(IOException e) {

e.printStackTrace();

}

### 2. ****Character Streams (****Reader ****/**** Writer****)****

* Used for reading and writing **character data** (text files).
* Base classes:
  + Reader (for reading characters)
  + Writer (for writing characters)

#### **Examples:**

FileWriter fw = new FileWriter("file.txt");

fw.write("Hello");

fw.close();

**try** {

FileReader fr = **new** FileReader("file.txt");

**int** ch = fr.read();

**while**(ch != -1) {

System.***out***.print((**char**)ch);

ch = fr.read();

}

fr.close();

}

**catch**(IOException e) {

e.printStackTrace();

}

### 3. ****Buffered Streams (Efficient I/O)****

* These wrap around other streams to add buffering and improve performance.
* Classes:
  + BufferedReader, BufferedWriter
  + BufferedInputStream, BufferedOutputStream

#### Example:

BufferedReader br = new BufferedReader(new FileReader("file.txt"));

String line = br.readLine();

br.close();

### 4. ****Data Streams (Read/Write Java primitives)****

* Used to read and write primitive Java data types (int, float, boolean, etc.)
* Classes:
  + DataInputStream, DataOutputStream

#### **Example:**

DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin"));

dos.writeInt(123);

dos.writeDouble(45.67);

dos.close();

### 5. ****Object Streams (Serialization)****

* Used to read and write **objects**.
* Classes:
  + ObjectInputStream, ObjectOutputStream

#### **Example:**

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("obj.ser"));

oos.writeObject(new MyClass());

oos.close();

### Summary Table

| **Stream Type** | **Read Class** | **Write Class** | **Data Type** |
| --- | --- | --- | --- |
| Byte Stream | InputStream | OutputStream | Binary |
| Character Stream | Reader | Writer | Text |
| Buffered Stream | BufferedReader | BufferedWriter | Text |
| Data Stream | DataInputStream | DataOutputStream | Primitives |
| Object Stream | ObjectInputStream | ObjectOutputStream | Objects |

### Why Use Streams?

* Handle I/O operations between programs and files, memory, or networks.
* Provide abstraction to work with different sources of data in a consistent way.
* Enables reading/writing in a sequential manner (one byte/character at a time).

Let's now explore two key areas:

## 1. ****Examples of Java I/O Streams (Traditional)****

Here are **examples** for each type of stream:

### A. ****Byte Stream Example**** – Reading a file byte by byte

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class ByteStreamExample {

public static void main(String[] args) throws IOException {

FileInputStream fis = new FileInputStream("input.bin");

FileOutputStream fos = new FileOutputStream("output.bin");

int data;

while ((data = fis.read()) != -1) {

fos.write(data); // copy byte

}

fis.close();

fos.close();

}

}

### B. ****Character Stream Example**** – Copy text from one file to another

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class CharacterStreamExample {

public static void main(String[] args) throws IOException {

FileReader fr = new FileReader("input.txt");

FileWriter fw = new FileWriter("output.txt");

int ch;

while ((ch = fr.read()) != -1) {

fw.write(ch);

}

fr.close();

fw.close();

}

}

### C. ****Buffered Stream Example**** – Read line by line

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class BufferedReaderExample {

public static void main(String[] args) throws IOException {

BufferedReader br = new BufferedReader(new FileReader("input.txt"));

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

br.close();

}

}

### D. ****Data Stream Example**** – Write and read primitives

import java.io.\*;

public class DataStreamExample {

public static void main(String[] args) throws IOException {

// Writing

DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin"));

dos.writeInt(100);

dos.writeDouble(99.99);

dos.writeBoolean(true);

dos.close();

// Reading

DataInputStream dis = new DataInputStream(new FileInputStream("data.bin"));

int num = dis.readInt();

double val = dis.readDouble();

boolean flag = dis.readBoolean();

dis.close();

System.out.println(num + " " + val + " " + flag);

}

}

### E. ****Object Stream Example**** – Serialization

import java.io.\*;

class Person implements Serializable {

String name;

int age;

Person(String name, int age) {

this.name = name; this.age = age;

}

}

public class ObjectStreamExample {

public static void main(String[] args) throws IOException, ClassNotFoundException {

// Serialization

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"));

oos.writeObject(new Person("John", 30));

oos.close();

// Deserialization

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"));

Person p = (Person) ois.readObject();

ois.close();

System.out.println(p.name + " - " + p.age);

}

}

## 2. ****Java 8 Stream API**** – Functional Programming with Collections

Java 8 introduced **Stream API** to operate on collections using functional style:

### Basic Syntax:

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

names.stream()

.filter(name -> name.startsWith("A"))

.sorted()

.forEach(System.out::println); // Output: Alice

### More Examples

#### **A. Convert list of strings to uppercase**

List<String> names = Arrays.asList("john", "alice", "bob");

List<String> upper = names.stream()

.map(String::toUpperCase)

.collect(Collectors.toList());

System.out.println(upper); // [JOHN, ALICE, BOB]

#### **B. Sum of even numbers in a list**

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

int sum = nums.stream()

.filter(n -> n % 2 == 0)

.mapToInt(Integer::intValue)

.sum();

System.out.println(sum); // 12

#### **C. Count names with more than 3 letters**

long count = names.stream()

.filter(name -> name.length() > 3)

.count();

System.out.println(count); // e.g., 2

### When to Use Stream API?

* When working with **collections** or **arrays**
* For **filtering, mapping, reducing, collecting**
* Cleaner and more **declarative** code

Here's a **real-life example** that combines:

* **Traditional I/O** (to read a text file)
* **Java 8 Stream API** (to filter and process lines)

## ****Goal:****

Read a file named employees.txt, filter employees with salary > 50000, and print their names.

### Sample employees.txt File:

Alice,55000

Bob,60000, Accounts

Carol,47000

David,70000

### Java Code: Combine I/O and Stream API

import java.io.\*;

import java.nio.file.\*;

import java.util.stream.\*;

public class EmployeeFilter {

public static void main(String[] args) {

try {

// Read lines from the file using Files.lines (Stream<String>)

Stream<String> lines = Files.lines(Paths.get("employees.txt"));

lines.map(line -> line.split(",")) // Split each line by comma

.filter(parts -> parts.length == 2) // Ensure proper format

.filter(parts -> Integer.parseInt(parts[1]) > 50000) // Salary > 50000

.map(parts -> parts[0]) // Get only the name

.forEach(System.out::println); // Print each name

lines.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

### Explanation:

| **Step** | **Action** |
| --- | --- |
| Files.lines(path) | Reads the file line by line as a Stream |
| .map(line -> split) | Splits each line into [name, salary] |
| .filter(salary > 50000) | Filters only employees with salary > 50000 |
| .map(parts[0]) | Extracts the name |
| .forEach(...) | Prints the result to the console |

### Output:

Alice

Bob

David

Here's an **enhanced version** of the previous example:

## ****Goal:****

* Read employees.txt
* Filter employees with salary > 50000
* Write the filtered data to high\_salary\_employees.txt

### Sample employees.txt

John,45000

Alice,55000

Bob,60000

Carol,47000

David,70000

### Java Code: Read → Filter → Write

import java.io.\*;

import java.nio.file.\*;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class FilterAndWriteEmployees {

public static void main(String[] args) {

Path inputPath = Paths.get("employees.txt");

Path outputPath = Paths.get("high\_salary\_employees.txt");

try (Stream<String> lines = Files.lines(inputPath)) {

// Process and collect filtered employees

List<String> filteredEmployees = lines

.map(line -> line.split(",")) // Split into name and salary

.filter(parts -> parts.length == 2) // Ensure format is correct

.filter(parts -> Integer.parseInt(parts[1]) > 50000)

.map(parts -> parts[0] + "," + parts[1]) // Rebuild the line

.collect(Collectors.toList());

// Write the filtered lines to the output file

Files.write(outputPath, filteredEmployees);

System.out.println("Filtered employees written to: " + outputPath.toString());

} catch (IOException e) {

e.printStackTrace();

}

}

}

### Output: high\_salary\_employees.txt

Alice,55000

Bob,60000

David,70000

### Summary of Key Methods Used:

| **Method** | **Purpose** |
| --- | --- |
| Files.lines(Path) | Read file as Stream |
| .split(",") | Split each line into name/salary |
| filter(...) | Filter based on salary condition |
| collect(Collectors.toList()) | Collect filtered lines into a list |
| Files.write(Path, List) | Write the list to a file |

Let's cover **enhancements** with the same employees.txt data.

## Sample employees.txt (again for reference)

John,45000

Alice,55000

Bob,60000

Carol,47000

David,70000

## 1. ****Sorting by Name or Salary****

### Sort by Name (Alphabetically)

// Sorted by name

List<String> sortedByName = filteredEmployees.stream()

.sorted((a, b) -> a.split(",")[0].compareToIgnoreCase(b.split(",")[0]))

.collect(Collectors.toList());

Files.write(Paths.get("sorted\_by\_name.txt"), sortedByName);

### Sort by Salary (Descending)

// Sorted by salary descending

List<String> sortedBySalary = filteredEmployees.stream()

.sorted((a, b) -> {

int salaryA = Integer.parseInt(a.split(",")[1]);

int salaryB = Integer.parseInt(b.split(",")[1]);

return Integer.compare(salaryB, salaryA); // descending

})

.collect(Collectors.toList());

Files.write(Paths.get("sorted\_by\_salary.txt"), sortedBySalary);

## 2. ****JSON Output****

### Define a Java Class Employee

class Employee {

String name;

int salary;

public Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

// Getters (required for JSON serialization)

public String getName() { return name; }

public int getSalary() { return salary; }

}

### Write to JSON (Using Jackson)

Add this Maven dependency to your pom.xml (or use a library like Gson):

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.17.0</version>

</dependency>

### Code to Write JSON File

import com.fasterxml.jackson.databind.ObjectMapper;

List<Employee> empList = filteredEmployees.stream()

.map(line -> {

String[] parts = line.split(",");

return new Employee(parts[0], Integer.parseInt(parts[1]));

})

.collect(Collectors.toList());

ObjectMapper mapper = new ObjectMapper();

mapper.writerWithDefaultPrettyPrinter().writeValue(new File("employees.json"), empList);

## 3. ****Writing as Serialized Objects****

### Serialize to a .ser File

import java.io.ObjectOutputStream;

import java.io.FileOutputStream;

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("employees.ser"))) {

oos.writeObject(empList); // empList must be Serializable

System.out.println("Serialized successfully.");

} catch (IOException e) {

e.printStackTrace();

}

Ensure Employee class implements Serializable:

class Employee implements Serializable {

String name;

int salary;

public Employee(String name, int salary) {

this.name = name; this.salary = salary;

}

}

## Deserializing from .ser File

import java.io.ObjectInputStream;

import java.io.FileInputStream;

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("employees.ser"))) {

List<Employee> deserializedList = (List<Employee>) ois.readObject();

deserializedList.forEach(emp ->

System.out.println(emp.name + " - " + emp.salary));

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

## Summary

| **Feature** | **Output File** | **Notes** |
| --- | --- | --- |
| Sort by Name | sorted\_by\_name.txt | Alphabetical order |
| Sort by Salary | sorted\_by\_salary.txt | Descending salary |
| JSON Output | employees.json | Jackson or Gson can be used |
| Object Serialization | employees.ser | Can be deserialized back to Java |

### Functional Interfaces in Java

A **Functional Interface** in Java is an interface that contains **only one abstract method**.

These are used primarily for **lambda expressions** and **method references**, introduced in Java 8.

### Key Points:

* It **may have** multiple **default or static methods**, but **only one abstract method**.
* Annotated with @FunctionalInterface (optional but helps the compiler ensure correctness).
* Commonly used with **Lambda Expressions**, **Streams**, and **Functional Programming** in Java.

### Syntax Example:

@FunctionalInterface

interface MyFunction {

void execute(); // Only one abstract method allowed

}

You can use it with a lambda:

public class Test {

public static void main(String[] args) {

MyFunction mf = () -> System.out.println("Hello from Lambda!");

mf.execute();

}

}

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

| **Interface** | **Method Signature** | **Description** |
| --- | --- | --- |
| Predicate<T> | boolean test(T t) | Used for conditional checks (returns boolean) |
| Function<T, R> | R apply(T t) | Transforms T to R |
| Consumer<T> | void accept(T t) | Consumes input without returning a result |
| Supplier<T> | T get() | Supplies a result of type T |
| UnaryOperator<T> | T apply(T t) | Same input and output types |
| BinaryOperator<T> | T apply(T t1, T t2) | Takes two inputs of same type, returns same type |

### Examples of Built-in Functional Interfaces

#### 1. Predicate Example:

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

Predicate<String> isEmpty = str -> str.isEmpty();

System.out.println(isEmpty.test("")); // true

System.out.println(isEmpty.test("Java")); // false

}

}

#### 2. Function Example:

import java.util.function.Function;

public class FunctionExample {

public static void main(String[] args) {

Function<String, Integer> lengthFunction = s -> s.length();

System.out.println(lengthFunction.apply("Lambda")); // Output: 6

}

}

#### 3. Consumer Example:

import java.util.function.Consumer;

public class ConsumerExample {

public static void main(String[] args) {

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

printUpper.accept("java"); // Output: JAVA

}

}

#### 4. Supplier Example:

import java.util.function.Supplier;

public class SupplierExample {

public static void main(String[] args) {

Supplier<Double> randomSupplier = () -> Math.random();

System.out.println(randomSupplier.get());

}

}

### Why Use Functional Interfaces?

* They allow concise and **readable code using lambdas**.
* They **enable functional programming style** in Java.
* Useful in **streams**, **collections processing**, and **event handling**.

### Custom Functional Interface Example:

@FunctionalInterface

interface MathOperation {

int operate(int a, int b);

}

public class LambdaMath {

public static void main(String[] args) {

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

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

}

}

### Optional Class in Java

The Optional class in Java is a **container object** used to **represent the presence or absence of a value**.

It was introduced in **Java 8** to help avoid NullPointerException.

### Package:

java.util.Optional

### Why Use Optional?

* Prevent null references.
* Write cleaner, safer code (no need to check for null explicitly).
* Helps in **functional programming** using map, filter, flatMap, etc.

### Creating Optional Instances

Optional<String> empty = Optional.empty(); // No value

Optional<String> name = Optional.of("John"); // Non-null value

Optional<String> nullable = Optional.ofNullable(null); // Can accept null

### Common Methods in Optional

| **Method** | **Description** |
| --- | --- |
| isPresent() | Returns true if value is present |
| ifPresent(Consumer<T>) | Executes code if value is present |
| orElse(T) | Returns value or default |
| orElseGet(Supplier<T>) | Returns value or result from supplier |
| orElseThrow() | Throws NoSuchElementException if no value is present |
| get() | Returns value (unsafe: throws exception if absent) |
| map(Function) | Transforms value if present |
| filter(Predicate) | Filters value using a predicate |
| flatMap(Function) | Like map, but avoids nested Optional<Optional<T>> |

### Examples

#### 1. Basic Usage

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

System.out.println(name.get()); // Output: Java

#### 2. Avoid NullPointerException

Optional<String> city = Optional.ofNullable(null);

System.out.println(city.isPresent()); // false

#### 3. Using orElse

String result = city.orElse("Unknown City");

System.out.println(result); // Output: Unknown City

#### 4. Using ifPresent

Optional<String> lang = Optional.of("Python");

lang.ifPresent(s -> System.out.println("Language: " + s));

#### 5. Using map

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

Optional<String> upper = name.map(String::toUpperCase);

System.out.println(upper.get()); // Output: JAVA

#### 6. Using filter

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

Optional<String> filtered = name.filter(n -> n.startsWith("J"));

System.out.println(filtered.isPresent()); // true

### Avoid This Pitfall

Optional<String> wrong = Optional.of(null); // ❌ Throws NullPointerException

Always use Optional.ofNullable(null) if there's a chance of null value.

### Practical Use Case: Return Optional from Method

public Optional<String> findUsernameById(int id) {

if (id == 1) return Optional.of("admin");

else return Optional.empty();

}

Optional<String> user = findUsernameById(1);

user.ifPresent(System.out::println);

**Default Methods in Interfaces – Java 8 Feature**

default methods allow interfaces to have **method implementations** without affecting classes that implement the interface.

**Why Introduced?**

Before Java 8:

* Interfaces could only have **abstract methods**.
* Adding a method to an interface would break all implementing classes.

With Java 8:

* default methods let you **add new methods to interfaces** without breaking existing code.

**Syntax:**

interface MyInterface {

void show(); // abstract method

default void greet() {

System.out.println("Hello from default method!");

}

}

**Usage in a Class:**

class MyClass implements MyInterface {

public void show() {

System.out.println("Implementing abstract method.");

}

}

public class Test {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.show(); // Output: Implementing abstract method.

obj.greet(); // Output: Hello from default method!

}

}

**Overriding Default Method**

class MyClass implements MyInterface {

public void show() {

System.out.println("Abstract method");

}

@Override

public void greet() {

System.out.println("Overridden default method");

}

}

**Default Method in Multiple Interfaces (Conflict Resolution)**

interface A {

default void hello() {

System.out.println("Hello from A");

}

}

interface B {

default void hello() {

System.out.println("Hello from B");

}

}

class C implements A, B {

@Override

public void hello() {

A.super.hello(); // You must resolve the conflict explicitly

}

}

**Summary Table:**

| **Feature** | **Java Version** | **Description** |
| --- | --- | --- |
| default method | Java 8 | Allows interface to provide method body |
| static method | Java 8 | Static methods in interfaces |
| Abstract method | All versions | Must be implemented by implementing class |

**Real-World Example: Java Collections**

List<String> list = Arrays.asList("a", "b", "c");

list.forEach(System.out::println); // `forEach` is a default method in Iterable