# **Java 8 features in detail**

**🔧 What is Functional Programming?**

Functional Programming is a programming paradigm where computation is treated as the evaluation of mathematical functions and avoids changing state or mutating data.

Key Characteristics:

* Functions are first-class citizens (they can be passed as arguments, returned from other functions)
* Immutability – data does not change
* Pure functions – same input always gives same output, with no side effects
* Declarative code – focus on what to do, not how
* Higher-order functions – functions that take functions as input or return them

**💡 Why Functional Programming was Introduced in Java?**

Java has been an object-oriented language since the beginning. However, modern development trends required:

1. More concise and expressive code

* Verbosity in Java (especially with anonymous classes) made functional tasks painful.

2. Support for parallelism & big data

* Functional programming fits perfectly with streams and lambda expressions, enabling parallel, lazy, and efficient data processing.

3. Trend in industry

* Other languages like Scala, Kotlin, JavaScript, and Python had embraced FP. Java had to evolve.

Java 8 introduced:

* Lambda expressions
* Functional interfaces (like Function<T, R>, Predicate<T>)
* Streams API
* Optional
* All aimed at functional-style coding!

**🎯 Purpose of Functional Programming in Java**

| **Purpose** | **Benefit** |
| --- | --- |
| Simplify collection/data processing | Streams + Lambdas |
| Encourage immutability | Fewer bugs, thread-safe |
| Enable parallelism | Easily scale across cores |
| Reduce boilerplate | Concise, readable code |
| Increase testability | Pure functions are easier to test |

**🌍 Real World Examples**

🔹 Example 1: Filter employees under 30

Before Java 8:  
List<Employee> young = new ArrayList<>();

for (Employee e : employees) {

if (e.getAge() < 30) {

young.add(e);

}

}

With FP (Java 8+):

List<Employee> young = employees.stream()

.filter(e -> e.getAge() < 30)

.collect(Collectors.toList());

✅ Cleaner, declarative, and no mutation!  
  
🔹 Example 2: Total salary of developers

double total = employees.stream()

.filter(e -> e.getRole().equals("developer"))

.mapToDouble(Employee::getSalary)

.sum();

🔹 Example 3: Passing functions (Higher-order function)

public static void process(List<String> names, Consumer<String> action) {

names.forEach(action);

}

// Call it like this:

process(List.of("Alice", "Bob"), name -> System.out.println(name.toUpperCase()));

🧠 Pure Function Example

public int square(int x) {

return x \* x;

}

* Always gives the same result
* No global state change
* No side effects

**🧪 Benefits in the Real World**

| **Benefit** | **How it helps** |
| --- | --- |
| **Parallel-friendly** | Pure functions don't rely on shared state |
| **Easier testing** | No hidden dependencies or state to mock |
| **More readable** | Focus on “what to do” rather than “how to do” |
| **Less error-prone** | Immutability avoids bugs due to state changes |

**🏁 Summary**

* **Functional Programming** in Java is **not a replacement** for OOP, but a **complement** to make code more readable, testable, and scalable.
* It’s enabled primarily via **lambdas, functional interfaces, and streams**.
* It’s great for **data processing**, **event-driven systems**, **multi-threaded applications**, and **cleaner domain logic**.

**🔍 Why were Streams introduced?**

1. **Verbose and error-prone loops:**
   * Before streams, developers had to use nested loops, temporary lists, and boilerplate code to filter, map, or transform data.
2. **Functional programming support:**
   * Java 8 introduced **lambda expressions** and **functional interfaces**. Streams leverage these to enable a more functional programming style.
3. **Efficiency through lazy evaluation:**
   * Stream operations are **lazily evaluated** – computation is only done when needed, which can lead to better performance.
4. **Parallelism made easy:**
   * Streams abstract away multi-threading. You can switch to parallel processing with .parallelStream().

**🎯 Purpose of Streams**

Streams allow you to:

* Filter data
* Transform data
* Aggregate results (like sum, average, max)
* Work with infinite sequences
* Chain multiple operations in a pipeline

**🔧 How Streams Work Internally**

**Stream Pipeline has 3 parts:**

1. **Source** – Where the data comes from (e.g., a List, Set, array).
2. **Intermediate operations** – Transformations like map, filter, sorted, which are **lazy**.
3. **Terminal operation** – Triggers the computation (e.g., collect, forEach, count).

✅ **Example Internal Working**:  
List<String> names = List.of("Alice", "Bob", "Charlie", "David");

List<String> result = names.stream()

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

.map(String::toUpperCase)

.collect(Collectors.toList());  
**Internally:**

* filter doesn’t do any actual work until collect is called.
* Elements flow through a **chain of operations** (called a **stream pipeline**).
* Only elements that match the filter pass through to map.
* The result is lazily built and finally collected into a list.

**🌍 Real World Example**

**Scenario: Processing employee data**

Imperative way (pre-Java 8):  
List<Employee> devs = new ArrayList<>();

for (Employee e : employees) {

if ("developer".equals(e.getRole()) && e.getAge() < 30) {

devs.add(e);

}

}  
  
Stream-based way:  
List<Employee> devs = employees.stream()

.filter(e -> "developer".equals(e.getRole()))

.filter(e -> e.getAge() < 30)

.collect(Collectors.toList());  
  
1] Cleaner, chainable filters

2] Easy to extend

3] Readable and declarative  
  
⚙️ Parallel Processing Example  
int sum = IntStream.range(1, 1\_000\_000)

.parallel()

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

.sum();

* Internally, Java uses **ForkJoinPool.commonPool()** to divide work among threads.
* Abstracts thread management while leveraging **multi-core CPUs**.

**🧠 Key Points to Remember**

| **Feature** | **Benefit** |
| --- | --- |
| **Declarative syntax** | Less code, more readability |
| **Lazy evaluation** | Efficient execution |
| **Functional style** | Easy to compose logic |
| **Parallelism** | Multithreading without complexity |
| **Immutability** | Safe operations without side-effects |

**🔹 1. Stream Source**

This is how a stream **originates** — from collections, arrays, I/O channels, etc.

**✅ Examples:**

**a) From a List**

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

Stream<String> nameStream = names.stream();

**b) From an Array**

int[] numbers = {1, 2, 3, 4};

IntStream numStream = Arrays.stream(numbers);

**c) From Files (Lines of text)**

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

**d) Infinite Streams**

Stream<Double> randoms = Stream.generate(Math::random);

Stream<Integer> counter = Stream.iterate(0, n -> n + 1);

**🔹 2. Intermediate Operations**

These transform or filter the stream.  
👉 They are **lazy** — nothing is processed until a **terminal** operation is triggered.

**🔁 Can be chained**

**✅ Common Intermediate Operations:**

**a) filter(Predicate)**

Filters elements based on a condition

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

names.stream()

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

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

**b) map(Function)**

Transforms each element

List<String> words = List.of("java", "stream");

words.stream()

.map(String::toUpperCase)

.forEach(System.out::println); // JAVA, STREAM

**c) flatMap(Function)**

Flattens nested structures

List<List<String>> nested = List.of(

List.of("a", "b"),

List.of("c", "d")

);

nested.stream()

.flatMap(List::stream)

.forEach(System.out::println); // a b c d

**d) distinct()**

Removes duplicates

Stream.of("java", "java", "stream")

.distinct()

.forEach(System.out::println); // java, stream

**e) sorted()**

Sorts elements (natural or with comparator)

List<String> fruits = List.of("Banana", "Apple", "Mango");

fruits.stream()

.sorted()

.forEach(System.out::println); // Apple, Banana, Mango

Or custom:

fruits.stream()

.sorted(Comparator.reverseOrder())

.forEach(System.out::println); // Mango, Banana, Apple

**f) limit(n) / skip(n)**

Take or skip elements

Stream.iterate(1, n -> n + 1)

.limit(5)

.forEach(System.out::println); // 1 2 3 4 5

List.of(10, 20, 30, 40, 50).stream()

.skip(2)

.forEach(System.out::println); // 30 40 50

**🔹 3. Terminal Operations**

These trigger the actual processing of the stream.

They either return a result or cause a side effect.  
  
**✅ Common Terminal Operations:**

**a) forEach(Consumer)**

Applies an action to each element (like printing)

List.of("Java", "Streams").stream()

.forEach(System.out::println);

**b) collect(Collector)**

Collects results into a collection or summary

List<String> upper = List.of("a", "b", "c").stream()

.map(String::toUpperCase)

.collect(Collectors.toList());

Or group by:

Map<Integer, List<String>> grouped = List.of("aa", "bbb", "c").stream()

.collect(Collectors.groupingBy(String::length));

**c) reduce()**

Reduces elements to a single result (like sum, concat)

int sum = List.of(1, 2, 3).stream()

.reduce(0, Integer::sum); // 6

Or concat:

String result = List.of("A", "B", "C").stream()

.reduce("", (a, b) -> a + b); // ABC

**d) count()**

Counts number of elements

long count = List.of(10, 20, 30).stream().count(); // 3

**e) min() / max()**

Finds min/max element using comparator

Optional<Integer> min = Stream.of(3, 5, 1).min(Integer::compareTo);

**f) anyMatch() / allMatch() / noneMatch()**

Check if some/all/none match a condition

boolean hasShort = List.of("Java", "Go", "Rust").stream()

.anyMatch(s -> s.length() <= 2); // true (Go)

**g) findFirst() / findAny()**

Retrieve one element (from sequential/parallel stream)

Optional<String> first = Stream.of("A", "B", "C").findFirst(); // A

**🧠 Final Notes**

| **Category** | **Examples** |
| --- | --- |
| **Source** | stream(), Arrays.stream(), Files.lines() |
| **Intermediate** | filter, map, flatMap, sorted, limit |
| **Terminal** | forEach, collect, reduce, count, min |

✅ **1. Filter even numbers from a list**

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

List<Integer> evens = numbers.stream()

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

.collect(Collectors.toList());  
  
✅ **2. Convert a list of strings to uppercase**

List<String> names = List.of("alice", "bob", "charlie");

List<String> upper = names.stream()

.map(String::toUpperCase)

.collect(Collectors.toList());  
  
**✅ 3. Get names of employees in a specific department**  
List<Employee> employees = getEmployeeList();

List<String> itNames = employees.stream()

.filter(e -> "IT".equals(e.getDept()))

.map(Employee::getName)

.collect(Collectors.toList());  
  
✅ **4. Count frequency of each word in a sentence**String sentence = "apple banana apple orange banana apple";

Map<String, Long> freq = Arrays.stream(sentence.split(" "))

.collect(Collectors.groupingBy(w -> w, Collectors.counting()));  
  
**✅ 5. Sum of salaries in a department**

double totalItSalary = employees.stream()

.filter(e -> "IT".equals(e.getDept()))

.mapToDouble(Employee::getSalary)

.sum();

**✅ 6. Group students by grade**

List<Student> students = getStudentList();

Map<String, List<Student>> byGrade = students.stream()

.collect(Collectors.groupingBy(Student::getGrade));

**✅ 7. Find the oldest person**

Optional<Person> oldest = people.stream()

.max(Comparator.comparing(Person::getAge));

**✅ 8. Find any product that is out of stock**

Optional<Product> outOfStock = products.stream()

.filter(p -> p.getQuantity() == 0)

.findAny();

**✅ 9. Join list of names with comma**

String joined = names.stream()

.collect(Collectors.joining(", "));

**✅ 10. Sort employees by salary descending**

List<Employee> sorted = employees.stream()

.sorted(Comparator.comparing(Employee::getSalary).reversed())

.collect(Collectors.toList());

**✅ 11. Partition people into adults and minors**

Map<Boolean, List<Person>> partitioned = people.stream()

.collect(Collectors.partitioningBy(p -> p.getAge() >= 18));

**✅ 12. Remove duplicates from a list**

List<String> uniqueNames = names.stream()

.distinct()

.collect(Collectors.toList());

**✅ 13. Get top 3 highest paid employees**

List<Employee> top3 = employees.stream()

.sorted(Comparator.comparing(Employee::getSalary).reversed())

.limit(3)

.collect(Collectors.toList());

**✅ 14. Flatten a list of lists (e.g. list of phone numbers)**

List<List<String>> phoneBook = List.of(

List.of("123", "456"),

List.of("789")

);

List<String> allNumbers = phoneBook.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

**✅ 15. Find the average age of all students**

double avgAge = students.stream()

.mapToInt(Student::getAge)

.average()

.orElse(0.0);

**✅ Java Streams Quiz**

**1. What type of operation is filter() in the Stream API?**

A. Source Operation  
B. Intermediate Operation  
C. Terminal Operation  
D. Reduction Operation

<details><summary>Answer</summary>B. Intermediate Operation</details>

**2. Which method is used to collect stream results into a list?**

A. .toList()  
B. .collect(Collectors.toList())  
C. .gather()  
D. .listify()

<details><summary>Answer</summary>B. `.collect(Collectors.toList())`</details>

**3. What is the result of this stream expression?**

Stream.of(1, 2, 3, 4)

.map(n -> n \* 2)

.filter(n -> n > 5)

.findFirst();

A. 2  
B. 4  
C. 6  
D. Optional.empty

<details><summary>Answer</summary>C. 6</details>

**4. Which of the following creates a stream from an array?**

A. Arrays.toStream()  
B. Stream.arrayOf()  
C. Arrays.stream()  
D. Stream.fromArray()

<details><summary>Answer</summary>C. `Arrays.stream()`</details>

**5. What does map() do in a stream pipeline?**

A. Filters elements  
B. Terminates the stream  
C. Transforms each element  
D. Skips elements

<details><summary>Answer</summary>C. Transforms each element</details>

**6. What is the output of this expression?**

List.of("a", "bb", "ccc")

.map(String::length)

.count();

A. 6  
B. 3  
C. Compilation error  
D. Runtime Exception

<details><summary>Answer</summary>C. Compilation error — `map()` can't be used directly on `List`, only on streams.</details>

**7. Which method ends the stream pipeline and returns a result?**

A. map()  
B. peek()  
C. filter()  
D. collect()

<details><summary>Answer</summary>D. `collect()`</details>

**8. Which operation is stateless and non-terminal?**

A. filter()  
B. collect()  
C. reduce()  
D. count()

<details><summary>Answer</summary>A. `filter()`</details>

**9. What will this code output?**

IntStream.range(1, 5).sum();

A. 10  
B. 15  
C. 5  
D. 0

<details><summary>Answer</summary>A. 10 (1 + 2 + 3 + 4)</details>

**10. Which stream operation is used to combine elements into a single result?**

A. collect()  
B. reduce()  
C. flatMap()  
D. sorted()

<details><summary>Answer</summary>B. `reduce()`</details>  
  
**✅ 1. IntStream**

**🔹 Common Use:**

Used when dealing with primitive int values without boxing into Integer.

**🔸 Key Methods:**

IntStream.range(int start, int end)

IntStream.rangeClosed(int start, int end) // inclusive

IntStream.of(int... values)

IntStream.generate(Supplier)

IntStream.sum()

IntStream.average()

IntStream.min(), max()

IntStream.map(), filter(), reduce()

IntStream.boxed() // converts to Stream<Integer>

**✅ Real-World Example:**

**Sum of all even numbers from 1 to 100**

int sum = IntStream.rangeClosed(1, 100)

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

.sum();

System.out.println("Sum: " + sum);

**✅ 2. LongStream**

**🔹 Common Use:**

Efficient handling of large integer sequences like timestamps, counters, or IDs.

**🔸 Key Methods:**

LongStream.of(long...)

LongStream.range(start, end)

LongStream.sum(), average(), min(), max()

LongStream.mapToInt(), mapToDouble(), boxed()

**✅ Real-World Example:**

**Generate 10 future timestamps spaced by 1 second**

LongStream.iterate(System.currentTimeMillis(), t -> t + 1000)

.limit(10)

.forEach(System.out::println);

**✅ 3. DoubleStream**

**🔹 Common Use:**

Used for floating-point calculations like prices, sensor data, or scientific values.

**🔸 Key Methods:**

DoubleStream.of(double...)

DoubleStream.generate(Supplier)

DoubleStream.sum(), average(), min(), max()

DoubleStream.mapToInt(), mapToLong(), boxed()

**✅ Real-World Example:**

**Convert temperatures from Celsius to Fahrenheit**

DoubleStream.of(36.6, 37.0, 38.2)

.map(c -> c \* 9/5 + 32)

.forEach(f -> System.out.println(f + " °F"));

**⚖️ When to Use Specialized Streams**

| **Use Case** | **Stream Type** |
| --- | --- |
| Working with integers or counters | IntStream |
| Timestamps, large IDs | LongStream |
| Prices, scientific data | DoubleStream |
| Want to avoid boxing | Specialized stream |
| Need to collect to List | Use .boxed() |

🚀 Performance Tip

Specialized streams significantly reduce memory overhead and CPU time by avoiding autoboxing (converting int to Integer, etc.), which can be costly in large-scale data processing.

**🧰 What Are Build Tools?**

Build tools automate:

* Compilation of code
* Packaging into JAR/WAR
* Dependency management
* Running tests
* Code analysis
* Deployment

**☕ Maven (Introduced in 2004)**

**📦 Key Concepts**

* **Convention over configuration**
* XML-based configuration (pom.xml)
* Follows a fixed project structure
* Uses **Central Repository** for dependencies

**🔧 Project Structure**

project/

├── pom.xml

└── src/

├── main/java

└── test/java

🧪 Basic pom.xml Example

<project xmlns="http://maven.apache.org/POM/4.0.0" ...>

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>demo-app</artifactId>

<version>1.0.0</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.3.20</version>

</dependency>

</dependencies>

</project>

**✅ Common Maven Commands**

* mvn clean – Clean compiled files
* mvn compile – Compile code
* mvn package – Create JAR/WAR
* mvn test – Run unit tests
* mvn install – Install to local repo

**⚙️ Gradle (Introduced in 2007)**

**📦 Key Concepts**

* **Convention + flexibility**
* Uses a **Groovy** or **Kotlin DSL** (build.gradle or build.gradle.kts)
* More performance-focused with **incremental builds** and **task caching**

**🧪 Basic build.gradle Example**

plugins {

id 'java'

}

group 'com.example'

version '1.0.0'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework:spring-core:5.3.20'

testImplementation 'junit:junit:4.13.2'

}

**✅ Common Gradle Commands**

* gradle clean – Clean build
* gradle build – Compile + test + package
* gradle test – Run tests
* gradle run – Run main class (with application plugin)

**⚔️ Gradle vs Maven Comparison**

| **Feature** | **Maven** | **Gradle** |
| --- | --- | --- |
| Configuration | XML (pom.xml) | Groovy/Kotlin (build.gradle) |
| Build Speed | Slower | Faster (Incremental, caching) |
| Dependency Management | Yes | Yes |
| Convention-based | Strict | Flexible |
| Plugin Ecosystem | Mature | Modern, extensible |
| IDE Support | Excellent | Excellent |
| Learning Curve | Easier for beginners | Slightly complex DSL |

**🚀 Real-World Example: Spring Boot Project**

**Maven**:

mvn archetype:generate -DgroupId=com.example -DartifactId=demo -DarchetypeArtifactId=maven-archetype-quickstart

**Gradle**:

gradle init --type java-application  
  
**Dates in java**  
  
**1. Shortcomings of the Old API**

1. **Mutability & Thread‑Safety**
   * java.util.Date and Calendar are mutable, so sharing instances across threads is unsafe.
2. **Poor API Design**
   * Months are zero‑based (Calendar.JANUARY == 0), years are offset from 1900, confusing getters (getYear() deprecated).
3. **Inconsistent Types**
   * Date actually holds date + time + timezone info ambiguously; formatting required DateFormat.
4. **Limited Functionality**
   * No built‑in support for periods (years/months/days) vs. durations (hours/minutes/seconds), no ISO‑8601 parsing by default.

**2. Goals of the Java 8 Date‑Time API**

* **Immutability** & **thread safety**
* Clear separation of **date**, **time**, **date‑time**, **instant**, **duration**, **period**
* **ISO‑8601**‑compliant by default
* Fluent, **method‑chainable** API
* Pluggable **formatting** & **parsing** via DateTimeFormatter

**3. Key Classes & Examples**

**3.1 LocalDate**

A date without time or timezone (e.g. 2025‑04‑17).

LocalDate today = LocalDate.now();

LocalDate birthday = LocalDate.of(1990, Month.JULY, 15);

LocalDate nextWeek = today.plusWeeks(1);

**3.2 LocalTime**

A time without date or timezone (e.g. 14:30:00).

LocalTime now = LocalTime.now();

LocalTime departure = LocalTime.parse("09:45:00");

LocalTime twoHoursLater = now.plusHours(2);

**3.3 LocalDateTime**

Date + time, no timezone (e.g. 2025‑04‑17T14:30).

LocalDateTime meeting = LocalDateTime.of(2025, 4, 17, 14, 30);

LocalDateTime formatted = LocalDateTime.parse("2025-04-17T14:30:00");

**3.4 ZonedDateTime**

Date + time + timezone (e.g. 2025‑04‑17T14:30+05:30[Asia/Kolkata]).

ZonedDateTime nowKolkata = ZonedDateTime.now(ZoneId.of("Asia/Kolkata"));

ZonedDateTime londonMeeting = ZonedDateTime.of(meeting, ZoneId.of("Europe/London"));

**3.5 Instant**

A point on the UTC timeline (wall‑clock vs. epoch).

Instant start = Instant.now();

Thread.sleep(500);

Instant end = Instant.now();

Duration elapsed = Duration.between(start, end);

**3.6 Duration vs. Period**

* **Duration**: time‑based (seconds, nanoseconds)
* **Period**: date‑based (years, months, days)

Duration twoHours = Duration.ofHours(2);

Period threeMonths = Period.ofMonths(3);

LocalDate threeMonthsLater = today.plus(threeMonths);

**3.7 DateTimeFormatter**

Immutable formatters for parsing/formatting.

DateTimeFormatter fmt = DateTimeFormatter.ofPattern("dd MMM uuuu");

String formatted = today.format(fmt); // "17 Apr 2025"

LocalDate parsed = LocalDate.parse("17 Apr 2025", fmt);

**3.8 Back‑and‑forth with Legacy API**

// Date → Instant → ZonedDateTime

Date legacyDate = new Date();

Instant instant = legacyDate.toInstant();

ZonedDateTime zdt = instant.atZone(ZoneId.systemDefault());

// ZonedDateTime → Instant → Date

Date backToDate = Date.from(zdt.toInstant());

**4. Why These Changes Matter**

* **Clarity**: Separate types for date vs. time vs. timestamp avoid confusion.
* **Safety**: Immutable objects eliminate thread‑safety issues.
* **Expressiveness**: Fluent API (e.g. .plusDays(5).minusHours(3)) is far more readable.
* **Standards**: Built‑in ISO‑8601 support means seamless interop (JSON, REST).
* **Flexibility**: Easy timezone conversions, precise durations, and periods.

**📘 Assignment: Employee Analytics Using Java Streams**

**🔍 Scenario:**

You are working in an HR tech startup. You’ve received a CSV file with thousands of employee records. Your task is to process this file and extract insights using **Java Streams API**.

**🗂️ Input File: employees.csv**

Each line in the CSV contains:

id,name,department,salary,location,age

📄 Example:

101,John Smith,Engineering,95000,New York,29

102,Alice Johnson,Marketing,75000,San Francisco,32

103,Bob Davis,Engineering,110000,Austin,35

104,Emma Watson,HR,60000,New York,26

**✅ Tasks to Implement Using Java Streams**

1. Load the CSV file and convert it into a list of Employee objects.
2. Find the average salary across all departments.
3. List all employees in the 'Engineering' department, sorted by salary descending.
4. Group employees by location and count how many are in each city.
5. Find the highest paid employee in the company.
6. Calculate the average age of employees per department.
7. Partition employees into two groups: younger than 30 and 30 or older.
8. Find departments with employees having salary > 100000.
9. Create a Map: Department -> List of Names, sorted alphabetically.

📦 **Class Definition to Use**

public class Employee {

private int id;

private String name;

private String department;

private double salary;

private String location;

private int age;

// Constructors, Getters, Setters, toString

}