JAVA PROGRAMMING LABORATORY

Book / LibraryManager, catalog maintained as an ArrayList<Book>

Add a new field to Book: private List<String> genres; (e.g. a book might have genres ["Programming", "Reference"]). Add a getter getGenres() and update your constructors to accept a List<String> genres parameter. This field is required for Part E.

Objective

To extend the Library Management System case study (Book / LibraryManager, catalog maintained as an ArrayList<Book>) into a fully modern Java implementation using generics (including bounded types and wildcards), lambda expressions, method references, custom functional interfaces, and the full range of Streams API operations — covering everything from basic filtering through flatMap, reduce, and summary statistics.

Recap: What You Already Have

This session assumes your Book class (bookId, title, author, price, isIssued, issuedTo) and LibraryManager (catalog as ArrayList<Book>) already exist, from the earlier labs. You will not rewrite Book, though Part A below asks you to ADD one new field to it. Everything else is new, built alongside what you already have.

Before starting, add a new field to Book: private List<String> genres; (e.g. a book might have genres ["Programming", "Reference"]). Add a getter getGenres() and update your constructors to accept a List<String> genres parameter. This field is required for Part E.

Part A — Generic Repository<T> and a Bounded Subclass

1. Repository<T>: wraps a private List<T> items. Provide add(T item), remove(T item), getAll(), size().
2. A generic method: T findFirst(Predicate<T> condition), returning the first match or null.
3. A generic method: void sortBy(Comparator<T> comparator), sorting the internal list in place.
4. Create a SECOND class NumericRepository<T extends Comparable<T>> that EXTENDS Repository<T> (this is a bounded type parameter — T must be Comparable). Add two methods: T findMax() and T findMin(), using the natural ordering (compareTo) of T directly — no Comparator needed, since the bound guarantees every T can compare itself to another T.
5. You cannot directly use NumericRepository<Book> unless Book implements Comparable<Book>. Make Book implement Comparable<Book>, ordering naturally by price. Confirm NumericRepository<Book> compiles and findMax()/findMin() return the correct books by price.

Part B — Lambda Expressions and Method References

Every item below must be demonstrated as BOTH a lambda expression AND, where possible, the equivalent method reference — print both to show they produce identical results.

1. Search: repository.findFirst(book -> book.getTitle().equals("Clean Code")) (lambda only — no direct method-reference equivalent for a custom condition like this).
2. Sort by title: as a lambda, (b1, b2) -> b1.getTitle().compareTo(b2.getTitle()); then again using the method reference Comparator.comparing(Book::getTitle). Confirm both produce the same order.
3. Sort by price: as a lambda, then again using Comparator.comparingDouble(Book::getPrice). Confirm both produce the same order.

4. Printing: iterate the catalog and print each book using catalog.forEach(book -> System.out.println(book)) as a lambda, then again using the method reference catalog.forEach(System.out::println). Confirm identical output.

Part C — Two Custom Functional Interfaces

FineCalculator — @FunctionalInterface interface FineCalculator { double calculate(Book book, int daysLate); }

1. Write TWO different lambda expressions assigned to FineCalculator variables: a flat rate per day, and a tiered rate that is higher for books priced above Rs. 1000.
2. Use each to compute a late fee for the same sample overdue book, and print both results for comparison.
3. BookValidator — a SECOND custom functional interface: @FunctionalInterface interface BookValidator { boolean isValid(Book book); }
4. Write at least two different lambda expressions assigned to BookValidator variables: one checking price > 0, another checking the title is non-empty AND the genres list is not empty.
5. Write a method boolean addIfValid(Repository<Book> repo, Book book, BookValidator validator) that only adds the book if validator.isValid(book) is true, and returns whether it was added. Test it with both a valid and a deliberately invalid book.

Part D — Core Streams API Operations

1. List<String> of titles of every ISSUED book (filter + map + collect).
2. A Map<String, List<Book>> grouping every book by author, using Collectors.groupingBy.
3. A Map<String, Long> counting how many books each author has, using Collectors.groupingBy with a downstream Collectors.counting().
4. The total catalog value using mapToDouble(...).sum().
5. The single most expensive book using max(Comparator.comparingDouble(...)).
6. A Map<Boolean, List<Book>> partitioning into "expensive" (>Rs.1000) and "affordable" books, using Collectors.partitioningBy.
7. A single summary String of all titles separated by ", ", using Collectors.joining(", ").

Part E — Advanced Streams: flatMap, reduce, toMap, Statistics

1. flatMap: produce a distinct, sorted List<String> of every genre appearing ANYWHERE in the catalog (each book has multiple genres, so you must flatten a Stream<List<String>> into a Stream<String> before collecting) — catalog.stream().flatMap(book -> book.getGenres().stream()).distinct().sorted().collect(...).
2. reduce: compute the total catalog value again, but this time using .reduce(0.0, (sum, book) -> sum + book.getPrice(), Double::sum) instead of mapToDouble().sum(). Confirm it matches Part D's answer exactly.
3. Collectors.toMap: build a Map<Integer, String> mapping each bookId directly to its title, using catalog.stream().collect(Collectors.toMap(Book::getBookId, Book::getTitle)).
4. Summary statistics: use catalog.stream().mapToDouble(Book::getPrice).summaryStatistics() to obtain count, sum, min, average, and max IN ONE CALL, and print all five values. Compare this against computing them separately in Part D
   — note in a comment which approach is more efficient and why.

Part F — Generic Wildcards

Wildcards let a method accept a generic type it doesn't need to know the EXACT type parameter of, only a bound on it.

1. Write a static method: static void printAll(List<? extends Book> books) that prints every book in any list of Book OR ANY SUBCLASS of Book. (If you don't have a Book subclass, briefly create a trivial one, e.g. class EBook extends Book, to prove this actually matters — printAll(List<Book>) alone would reject a List<EBook> without the wildcard.)
2. Write a static method: static double sumPrices(List<? extends Book> books) using a wildcard, and explain in a comment why ? extends Book is correct here but List<Book> would be too restrictive.
3. Write a generic class Pair<K, V> with two fields of different type parameters, a constructor, and getters getKey()/getValue(). Create a Pair<String, Double> representing ("Most expensive book title", price) from your Part D answer, and print it.

Part G — Integration Test Driver and Written Analysis

1. Populate the catalog with AT LEAST 15 books across AT LEAST 4 different authors, a range of prices (some below, some above Rs. 1000), and overlapping genres across books.
2. Call and print the output of every method from Parts A through F, clearly labelled with which Part/method produced each block of output.
3. Deliberately call findFirst() with a Predicate matching nothing, and confirm the null is handled without crashing.
4. Deliberately call summaryStatistics() and toMap() on the FULL, unfiltered catalog, and separately on a filtered (e.g. only "Programming" genre) sub-list, to show the same stream operations composing with an earlier filter() step.

Written analysis (answer in 5–10 sentences total, submitted as comments at the end of your Main class or a separate text file):

- When would you prefer a method reference over a lambda expression, and when would you have no choice but to use a lambda?
- Why does NumericRepository<T extends Comparable<T>> need the bound at all — what would break if T had no bound and you tried to call compareTo() inside findMax()?
- Why is List<? extends Book> the correct wildcard for a method that only READS from the list, while a method that needs to ADD Book objects to the list could not safely use ? extends?
- Between mapToDouble(...).sum() and summaryStatistics(), which would you choose if you needed count, sum, AND average together, and why?

What This Session Tests

- Generic classes AND bounded generic type parameters (T extends Comparable<T>)
- Generic methods that accept functional interfaces parameterised by the same type
- Lambda expressions vs. their equivalent method references, and recognising when each applies
- Writing and using MULTIPLE custom functional interfaces, not just the built-in ones
- The full Streams API toolkit: filter, map, sorted, collect, groupingBy (with downstream collectors), partitioningBy, joining, flatMap, reduce, toMap, and summaryStatistics
- Generic wildcards (? extends) and why they exist as distinct from a plain generic parameter
- Composing stream operations (filtering before collecting statistics) rather than treating each as an isolated example