Lab Guide for Core Java, Git, Build Tools & JUnit Testing

Lab 1: Bank Account System

Implement deposits, withdrawals, and balance check using classes & methods.

Objective

Build a console app in **Java** that models a bank account using **classes and methods**.
You'll implement:

- `deposit(double amount)`
- `withdraw(double amount)`
- `getBalance()`

Step 1 — Create a New Project in IntelliJ IDEA CE

1. Open **IntelliJ IDEA CE**.
2. **File → New → Project → Java**
   - Select your **JDK 21** (or higher).
   - Project name: `BankAccountLab`
   - Finish.
3. In the **Project window**, right-click `src` → **New → Java Class**.

Step 2 — Implement the Domain Class

Create `BankAccount.java`:

```
public class BankAccount {
    private final String accountNumber;
    private final String holderName;
    private double balance;

    public BankAccount(String accountNumber, String holderName,
double openingBalance) {
        this.accountNumber = accountNumber;
        this.holderName = holderName;
        this.balance = openingBalance;
    }

    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Deposit amount must be > 0");
        }else{
            balance += amount;
```

```
        }
    }

    public void withdraw(double amount){
        if (amount <= 0) {
            System.out.println("Withdrawal amount must be > 0");
        }else if (balance < amount) {
            System.out.println("Insufficient funds: balance=" +
balance + ", requested=" + amount);
        }else{
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

Step 3 — Console Menu (Main App)

Create `BankApp.java`:

```
import java.util.Scanner;

public class BankApp {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Create one account for demo
        BankAccount account = new BankAccount("ACC1001", "Alex",
500.0);

        while (true) {
            System.out.println("\n=== Bank Menu ===");
            System.out.println("1) Deposit");
            System.out.println("2) Withdraw");
            System.out.println("3) Check Balance");
            System.out.println("4) Exit");
            System.out.print("Choose: ");

            int choice = sc.nextInt();
                switch (choice) {
                    case 1 -> {
                        System.out.print("Enter deposit amount:
");
                        double amt = sc.nextDouble();
                        account.deposit(amt);
                    }
                    case 2 -> {
                        System.out.print("Enter withdrawal
amount: ");
                        double amt = sc.nextDouble();
```

```
                        account.withdraw(amt);
                    }
                    case 3 -> System.out.println("Current
balance: " + account.getBalance());
                    case 4 -> {
                        System.out.println("Goodbye!");
                        return;
                    }
                    default -> System.out.println("Invalid
choice. Try 1-4.");
                }

        }
    }

}
```

Step 4 — Run in IntelliJ

1. In the **Project Explorer**, right-click `BankApp.java`.
2. Select **Run 'BankApp.main()'**.
3. Test menu options:
    - Deposit `200` → balance updates.
    - Withdraw `100` → balance updates.
    - Withdraw more than balance → error message.
    - Enter negative deposit/withdraw → error message.

Lab 2: Employee Class Hierarchy

Create base class Employee and subclasses (Manager, Developer) demonstrating inheritance & polymorphism.

Objective

Create a base class `Employee` and two subclasses (`Manager`, `Developer`) to demonstrate **inheritance** and **polymorphism**. Implement shared behavior in the base class and override/extend behavior in subclasses. Run a short program that treats subclass instances as `Employee` references and shows runtime method dispatch.

Design (classes & responsibilities)

- `Employee` (abstract) — id, name, salary; common getters/setters; default `calculateBonus()`; abstract `work()`; `getDetails()`.
- `Manager` (extends Employee) — extra `teamSize`; overrides `work()` and `calculateBonus()`.
- `Developer` (extends Employee) — extra `primaryLanguage`; overrides `work()` and `calculateBonus()`.

Step 1 — Create a new Java project

IntelliJ IDEA CE

1. File → New → Project → Java → choose JDK 17/21 → Project name: `EmployeeHierarchyLab`.
2. Right-click `src` → New → Java Class → create the classes below.

Step 2 — `Employee.java` (base class)

```
// src/Employee.java
public abstract class Employee {
    private final int id;
    private final String name;
    private double salary;

    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public double getSalary() { return salary; }
    public void setSalary(double salary) {
        this.salary = salary;
    }
```

```
    /**
     * Default bonus calculation: 5% of salary.
     * Subclasses may override.
     */
    public double calculateBonus() {
        return salary * 0.05;
    }

    /** Concrete classes must implement actual work behavior. */
    public abstract void work();

    public String getDetails() {
        return String.format("%s[id=%d,name=%s,salary=%.2f]",
                this.getClass().getSimpleName(), id, name,
salary);
    }
}
```

Step 3 — Manager.java

```
// src/Manager.java
public class Manager extends Employee {
    private int teamSize;

    public Manager(int id, String name, double salary, int
teamSize) {
        super(id, name, salary);
        this.teamSize = teamSize;
    }

    public int getTeamSize() { return teamSize; }
    public void setTeamSize(int teamSize) { this.teamSize =
teamSize; }

    @Override
    public void work() {
        System.out.println(getName() + " (Manager) is planning
strategy and leading a team of " + teamSize);
    }

    @Override
    public double calculateBonus() {
        // Managers: 10% of salary + flat amount per team member
        return getSalary() * 0.10 + teamSize * 100.0;
    }
}
```

Step 4 — Developer.java

```
// src/Developer.java
public class Developer extends Employee {
    private final String primaryLanguage;
```

```
    public Developer(int id, String name, double salary, String
primaryLanguage) {
        super(id, name, salary);
        this.primaryLanguage = primaryLanguage;
    }

    public String getPrimaryLanguage() { return primaryLanguage;
}

    @Override
    public void work() {
        System.out.println(getName() + " (Developer) is coding in
" + primaryLanguage);
    }

    @Override
    public double calculateBonus() {
        // Developers: 7% base bonus; extra fixed bonus for Java
devs
        double bonus = getSalary() * 0.07;
        if ("Java".equalsIgnoreCase(primaryLanguage)) bonus +=
200.0;
        return bonus;
    }
}
```

Step 5 — `MainApp.java` — demonstrate inheritance & polymorphism

```
// src/MainApp.java
```

```
public class MainApp {
    public static void main(String[] args) {
        Employee[] employees = new Employee[3];
        employees[0] = new Manager(1, "Priya", 120000.00, 5);
        employees[1] = new Developer(2, "Rahul", 90000.00,
"Java");
        employees[2] = new Developer(3, "Nisha", 85000.00,
"JavaScript");

        System.out.println("=== Team Work & Bonuses ===");
        for (Employee e : employees) {
            // runtime (dynamic) dispatch: correct override will
be invoked
            e.work();
            System.out.printf("%s -> bonus=%.2f%n",
e.getDetails(), e.calculateBonus());
        }

        System.out.println("\n=== Promote Java developers by 10%
(example of instanceof) ===");
        for (Employee e : employees) {
            if (e instanceof Developer dev &&
```

```
"Java".equalsIgnoreCase(dev.getPrimaryLanguage())) {
                double old = dev.getSalary();
                dev.setSalary(old * 1.10);
                System.out.printf("Promoted %s: salary %.2f ->
%.2f%n", dev.getName(), old, dev.getSalary());
            }
        }

        System.out.println("\n=== After Promotion: Bonuses
recomputed ===");
        for (Employee e : employees) {
            System.out.printf("%s -> bonus=%.2f%n",
e.getDetails(), e.calculateBonus());
        }
    }
}
```

Step 6 — Run & expected output

Run in IntelliJ: right-click `MainApp` → Run `MainApp.main()`

Sample output

```
=== Team Work & Bonuses ===
Priya (Manager) is planning strategy and leading a team of 5
Manager[id=1,name=Priya,salary=120000.00] -> bonus=12200.00
Rahul (Developer) is coding in Java
Developer[id=2,name=Rahul,salary=90000.00] -> bonus=6300.00
Nisha (Developer) is coding in JavaScript
Developer[id=3,name=Nisha,salary=85000.00] -> bonus=5950.00

=== Promote Java developers by 10% (example of instanceof) ===
Promoted Rahul: salary 90000.00 -> 99000.00

=== After Promotion: Bonuses recomputed ===
Manager[id=1,name=Priya,salary=120000.00] -> bonus=12200.00
Developer[id=2,name=Rahul,salary=99000.00] -> bonus=7130.00
Developer[id=3,name=Nisha,salary=85000.00] -> bonus=5950.00
```

Lab 3: Debugging in IDE

Set breakpoints, inspect variables, step into/step over methods

Objective

Learn how to debug a Java application in **IntelliJ IDEA Community Edition** by setting breakpoints, inspecting variables, and using the step controls (step-into / step-over / step-out). You'll practice finding a bug (exception) and using conditional breakpoints, watches, and Evaluate Expression to diagnose and fix it.

Sample project

Calculator.java

```
// src/Calculator.java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int divide(int a, int b) {
        return a / b;
    }

    public int compute(int x) {
        int sum = add(x, 2);           // -> step into add()
        int result = divide(sum, x-1); // -> may throw when x == 1
        return result * 2;
    }
}
```

DebugDemo.java

```
// src/DebugDemo.java
public class DebugDemo {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        for (int i = 3; i >= 1; i--) {
            System.out.println("i = " + i + " => result = " +
calc.compute(i));
        }
    }
}
```

1) Create the project & add files

- File → New → Project → Java. Name it `DebuggingLab`.
- Add `Calculator.java` and `DebugDemo.java` under `src/`.

2) Set a simple breakpoint

- Open `DebugDemo.java`. Click the **left gutter** (left of the line numbers) on the line inside the loop (e.g., on the `System.out.println(...)` line) — a red dot appears.
- (Alternate) place a breakpoint on the `int result = divide(...)` line in `Calculator.compute()` to step into the method.

3) Start Debug mode

- Run → Debug 'DebugDemo' (or click the green **bug** icon).
- The program will start and **suspend** at your breakpoint. The Debug tool window appears.

4) Inspect variables & frames

- In the **Variables** panel (Debug tool window) you can see local variables (e.g., `i`, `calc`, method locals).
- Hover a variable in the editor to see its value inline.
- The **Frames** pane shows the call stack; click higher/lower to inspect different stack frames.

5) Step controls (observe runtime dispatch)

- **Step Over**: moves to the next line in the current method (does not enter method calls). Useful when you trust the called method.
- **Step Into**: enters the called method so you can debug it line-by-line.
- **Step Out**: finish current method and return to caller.
- **Resume**: continue until next breakpoint or program end.
  Use these to step from `DebugDemo.main()` into `Calculator.compute()` and from there **into** `divide()`.

6) Conditional breakpoint

- Right-click a breakpoint → **More** (or click the gear) → **Condition...**
- Enter a condition like `i == 1` or `sum > 4` so the breakpoint only suspends when the condition is true. This is handy in loops.

Common default shortcuts (defaults — check IntelliJ keymap for your platform)

- Toggle breakpoint: **Ctrl+F8** (Win/Linux) / **⌘F8** (mac)
- Step Over: **F8** / **F8**
- Step Into: **F7** / **F7**
- Step Out: **Shift+F8** / **Shift+F8**
- Resume Program: **F9** / **F9**

- Evaluate Expression: **Alt+F8** (Win) / **Option+F8** (mac)
  (If a shortcut differs on your machine, the Run or Debug menu shows the platform shortcut next to the action.)

Lab 4: Library Management System with Custom Exception

- Store books in a `List`, search/filter using Streams API.
- Create `InvalidBookException` for invalid data input.

Objective

Build a small console app that stores `Book` objects in a `List`, and uses the **Streams API** to search & filter books. Also create a custom checked exception `InvalidBookException` which is thrown when invalid book data is supplied.

Step 1 — Project setup (IntelliJ)

1. Open IntelliJ IDEA CE → **File → New → Project → Java**. Name: `LibraryLab`. Choose JDK 21.
2. Right-click `src` → **New → Java Class** — create the classes below.

Step 2 — Create the custom exception

`InvalidBookException.java`

```
public class InvalidBookException extends Exception {
    public InvalidBookException(String message) {
        super(message);
    }
}
```

This is a checked exception so callers must handle or declare it. It makes validation explicit in constructors or APIs.

Step 3 — Create the Book class (with validation)

`Book.java`

```
import java.util.Objects;

public class Book {
    private final String id;
    private final String title;
    private final String author;
    private final int year;
    private final String genre;
    private boolean available;

    public Book(String id, String title, String author, int year,
String genre) throws InvalidBookException {
        if (id == null || id.isBlank()) throw new
```

```
InvalidBookException("id is required");
        if (title == null || title.isBlank()) throw new
InvalidBookException("title is required");
        if (author == null || author.isBlank()) throw new
InvalidBookException("author is required");
        if (year < 0) throw new InvalidBookException("year must
be >= 0");
        if (genre == null || genre.isBlank()) throw new
InvalidBookException("genre is required");

        this.id = id;
        this.title = title;
        this.author = author;
        this.year = year;
        this.genre = genre;
        this.available = true;
    }

    // getters
    public String getId() { return id; }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public int getYear() { return year; }
    public String getGenre() { return genre; }
    public boolean isAvailable() { return available; }

    // mutable availability
    public void setAvailable(boolean available) { this.available
= available; }

    @Override
    public String toString() {
        return
String.format("Book[id=%s,title=%s,author=%s,year=%d,genre=%s,ava
ilable=%s]",
                id, title, author, year, genre, available);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Book)) return false;
        Book book = (Book) o;
        return Objects.equals(id, book.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
```

Step 4 — Library class: store books and Streams API methods

Library.java

```
import java.util.*;
import java.util.stream.Collectors;

public class Library {
    private final List<Book> books = new ArrayList<>();

    // add a book (Book constructor already validates and may
throw)
    public void addBook(Book book) {
        // prevent duplicates by id
        boolean exists = books.stream().anyMatch(b ->
b.getId().equals(book.getId()));
        if (!exists) books.add(book);
    }

    public boolean removeBookById(String id) {
        return books.removeIf(b -> b.getId().equals(id));
    }

    // find books where title contains query (case-insensitive)
    public List<Book> searchByTitle(String query) {
        String q = query == null ? "" : query.toLowerCase();
        return books.stream()
                .filter(b ->
b.getTitle().toLowerCase().contains(q))
                .collect(Collectors.toList());
    }

    public List<Book> searchByAuthor(String authorQuery) {
        String q = authorQuery == null ? "" :
authorQuery.toLowerCase();
        return books.stream()
                .filter(b ->
b.getAuthor().toLowerCase().contains(q))
                .collect(Collectors.toList());
    }

    // filter by inclusive year range
    public List<Book> filterByYearRange(int fromYear, int toYear)
{
        return books.stream()
                .filter(b -> b.getYear() >= fromYear &&
b.getYear() <= toYear)
                .collect(Collectors.toList());
    }

    public List<Book> filterByGenre(String genre) {
        String g = genre == null ? "" : genre.toLowerCase();
```

```
        return books.stream()
                .filter(b ->
b.getGenre().toLowerCase().equals(g))
                .collect(Collectors.toList());
    }

    public List<Book> listAvailableBooks() {
        return books.stream()
                .filter(Book::isAvailable)
                .collect(Collectors.toList());
    }

    public Optional<Book> findById(String id) {
        return books.stream().filter(b ->
b.getId().equals(id)).findFirst();
    }

    public boolean borrowBook(String id) {
        Optional<Book> opt = findById(id);
        if (opt.isEmpty()) return false;
        Book b = opt.get();
        if (!b.isAvailable()) return false;
        b.setAvailable(false);
        return true;
    }

    public boolean returnBook(String id) {
        Optional<Book> opt = findById(id);
        if (opt.isEmpty()) return false;
        Book b = opt.get();
        if (b.isAvailable()) return false;
        b.setAvailable(true);
        return true;
    }

    // Sorting example
    public List<Book> getAllBooksSortedByTitle() {
        return books.stream()
                .sorted(Comparator.comparing(Book::getTitle,
String.CASE_INSENSITIVE_ORDER))
                .collect(Collectors.toList());
    }

    // Grouping example
    public Map<String, List<Book>> groupByGenre() {
        return
books.stream().collect(Collectors.groupingBy(Book::getGenre));
    }

    // return a copy of internal list (defensive)
    public List<Book> getAllBooks() {
        return new ArrayList<>(books);
```

```
    }
}
```

Step 5 — LibraryApp.java — demonstrate Streams usage

LibraryApp.java

```
import java.util.List;
import java.util.Map;

public class LibraryApp {
    public static void main(String[] args) {
        Library lib = new Library();

        // Add sample books (handle InvalidBookException)
        try {
            lib.addBook(new Book("B1", "Effective Java", "Joshua
Bloch", 2018, "Programming"));
            lib.addBook(new Book("B2", "Clean Code", "Robert C.
Martin", 2008, "Programming"));
            lib.addBook(new Book("B3", "The Pragmatic
Programmer", "Andrew Hunt", 1999, "Programming"));
            lib.addBook(new Book("B4", "The Hobbit", "J.R.R.
Tolkien", 1937, "Fantasy"));
            lib.addBook(new Book("B5", "Harry Potter and the
Sorcerer's Stone", "J.K. Rowling", 1997, "Fantasy"));
        } catch (InvalidBookException e) {
            System.err.println("Failed to create a book: " +
e.getMessage());
        }

        System.out.println("=== All Books Sorted By Title ===");

lib.getAllBooksSortedByTitle().forEach(System.out::println);

        System.out.println("\n=== Search title contains 'code'
===");
        List<Book> codeBooks = lib.searchByTitle("code");
        codeBooks.forEach(System.out::println);

        System.out.println("\n=== Programming books (filter by
genre) ===");

lib.filterByGenre("Programming").forEach(System.out::println);

        System.out.println("\n=== Books published between 1990
and 2010 ===");
        lib.filterByYearRange(1990,
2010).forEach(System.out::println);

        System.out.println("\n=== Borrow B2 (Clean Code) ===");
        boolean borrowed = lib.borrowBook("B2");
        System.out.println("Borrowed B2? " + borrowed);
```

```
        System.out.println("Available books now:");
        lib.listAvailableBooks().forEach(System.out::println);

        System.out.println("\n=== Group books by genre ===");
        Map<String, List<Book>> grouped = lib.groupByGenre();
        grouped.forEach((genre, list) -> {
            System.out.println(genre + " -> " + list.size() + "
book(s)");
        });

        // show invalid book attempt (example)
        try {
            Book bad = new Book("", "", "", -1, "");
            lib.addBook(bad);
        } catch (InvalidBookException e) {
            System.out.println("\nAttempt to add invalid book
failed: " + e.getMessage());
        }
    }
}
```

Step 6 — Run it

In IntelliJ: Right-click `LibraryApp.java` → Run `LibraryApp.main()`

Sample output:

```
=== All Books Sorted By Title ===
Book[id=B2,title=Clean Code,author=Robert C.
Martin,year=2008,genre=Programming,available=true]
Book[id=B1,title=Effective Java,author=Joshua
Bloch,year=2018,genre=Programming,available=true]
Book[id=B5,title=Harry Potter and the Sorcerer's Stone,author=J.K.
Rowling,year=1997,genre=Fantasy,available=true]
Book[id=B4,title=The Hobbit,author=J.R.R.
Tolkien,year=1937,genre=Fantasy,available=true]
Book[id=B3,title=The Pragmatic Programmer,author=Andrew
Hunt,year=1999,genre=Programming,available=true]

=== Search title contains 'code' ===
Book[id=B2,title=Clean Code,author=Robert C.
Martin,year=2008,genre=Programming,available=true]

=== Programming books (filter by genre) ===
Book[id=B1,title=Effective Java,author=Joshua
Bloch,year=2018,genre=Programming,available=true]
Book[id=B2,title=Clean Code,author=Robert C.
Martin,year=2008,genre=Programming,available=true]
Book[id=B3,title=The Pragmatic Programmer,author=Andrew
Hunt,year=1999,genre=Programming,available=true]

=== Books published between 1990 and 2010 ===
Book[id=B2,title=Clean Code,author=Robert C.
Martin,year=2008,genre=Programming,available=true]
```

```
Book[id=B3,title=The Pragmatic Programmer,author=Andrew
Hunt,year=1999,genre=Programming,available=true]
Book[id=B5,title=Harry Potter and the Sorcerer's Stone,author=J.K.
Rowling,year=1997,genre=Fantasy,available=true]

=== Borrow B2 (Clean Code) ===
Borrowed B2? true
Available books now:
Book[id=B1,title=Effective Java,author=Joshua
Bloch,year=2018,genre=Programming,available=true]
Book[id=B3,title=The Pragmatic Programmer,author=Andrew
Hunt,year=1999,genre=Programming,available=true]
Book[id=B4,title=The Hobbit,author=J.R.R.
Tolkien,year=1937,genre=Fantasy,available=true]
Book[id=B5,title=Harry Potter and the Sorcerer's Stone,author=J.K.
Rowling,year=1997,genre=Fantasy,available=true]

=== Group books by genre ===
Fantasy -> 2 book(s)
Programming -> 3 book(s)

Attempt to add invalid book failed: id is required
```

Lab 5: Multi-threaded Ticket Booking System

- Multiple threads booking seats.

Objective

Understand how **synchronization** ensures thread safety in a multi-threaded program by implementing a simple **ticket booking system** where multiple threads try to book the same seat.

Step 1 — Create a Java project in IntelliJ IDEA CE

1. Open **IntelliJ IDEA Community Edition**.
2. Go to **File → New → Project → Java →** Name it `TicketBookingSyncLab`.
3. Add the classes below inside the `src` folder.

Step 2 — Code

TicketService.java

```
public class TicketService {
    private boolean[] seats;

    public TicketService(int capacity) {
        seats = new boolean[capacity + 1]; // index 1..capacity
        for (int i = 1; i <= capacity; i++) {
            seats[i] = true; // true = available
        }
    }

    // synchronized method to prevent race condition
    public synchronized boolean bookSeat(int seat, String user) {
        if (seat < 1 || seat >= seats.length) {
            System.out.println(user + " tried invalid seat " +
seat);
            return false;
        }
        if (!seats[seat]) {
            System.out.println(user + " failed, seat " + seat + "
already booked.");
            return false;
        }
        seats[seat] = false;
        System.out.println(user + " successfully booked seat " +
seat);
        return true;
    }
}
```

BookingTask.java

```
public class BookingTask implements Runnable {
    private final TicketService service;
    private final int seat;
    private final String user;

    public BookingTask(TicketService service, int seat, String
user) {
        this.service = service;
        this.seat = seat;
        this.user = user;
    }

    @Override
    public void run() {
        service.bookSeat(seat, user);
    }
}
```

BookingApp.java

```
public class BookingApp {
    public static void main(String[] args) {
        TicketService service = new TicketService(5); // 5 seats
total

        int targetSeat = 1; // all threads try seat 1
        for (int i = 1; i <= 10; i++) {
            String user = "User-" + i;
            Thread t = new Thread(new BookingTask(service,
targetSeat, user));
            t.start();
        }
    }
}
```

Step 3 — Run the program

1. Right-click `BookingApp.java` → **Run 'BookingApp.main()'**.
2. Observe output:
    - With `synchronized`, only **one user** should succeed in booking seat `1`.
    - Other users will see `"failed, seat 1 already booked."`.

Sample output

```
User-1 successfully booked seat 1
User-10 failed, seat 1 already booked.
User-9 failed, seat 1 already booked.
User-8 failed, seat 1 already booked.
User-7 failed, seat 1 already booked.
User-6 failed, seat 1 already booked.
User-5 failed, seat 1 already booked.
```

```
User-4 failed, seat 1 already booked.
User-3 failed, seat 1 already booked.
User-2 failed, seat 1 already booked.
```

Step 4 — Exercises

1. Remove `synchronized` from `bookSeat` and run again — you may see multiple users booking the same seat (race condition).
2. Change `targetSeat` so each thread books a different seat; confirm all succeed.
3. Increase threads to 100 to see synchronization effect more clearly.

Lab 6: Employee Records with Stream API

- Filter employees by salary, department, etc.

Objective

Build a small Java console app that stores `Employee` records in a `List` and uses the **Streams API** to filter, sort, group and aggregate by salary, department, name, etc. You'll practice common stream patterns (`filter`, `map`, `collect`) and return results in useful shapes (lists, maps).

Step 1 — Create the project

In IntelliJ IDEA CE: File → New → Project → Java → name it `EmployeeStreamsLab`. Create `src/` and add the classes below (no packages for simplicity).

Step 2 — `Employee.java` (POJO)

```
import java.time.LocalDate;
import java.util.Objects;

public class Employee {
    private final int id;
    private final String name;
    private final String department;
    private double salary;
    private final LocalDate joiningDate;

    public Employee(int id, String name, String department,
double salary, LocalDate joiningDate) {
        if (name == null || name.isBlank()) throw new
IllegalArgumentException("name required");
        if (department == null || department.isBlank()) throw new
IllegalArgumentException("department required");
        if (salary < 0) throw new
IllegalArgumentException("salary must be >= 0");

        this.id = id;
        this.name = name;
        this.department = department;
        this.salary = salary;
        this.joiningDate = joiningDate == null ? LocalDate.now()
: joiningDate;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public String getDepartment() { return department; }
    public double getSalary() { return salary; }
    public void setSalary(double salary) { this.salary = salary;
}
```

```
    public LocalDate getJoiningDate() { return joiningDate; }

    @Override
    public String toString() {
        return
String.format("Employee[id=%d,name=%s,dept=%s,salary=%.2f,joined=
%s]",
                id, name, department, salary, joiningDate);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Employee)) return false;
        Employee that = (Employee) o;
        return id == that.id;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
```

Step 3 — `EmployeeService.java` (Streams methods)

```
import java.util.*;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class EmployeeService {
    private final List<Employee> employees = new ArrayList<>();

    public void add(Employee e) { employees.add(e); }
    public List<Employee> getAll() { return new
ArrayList<>(employees); }

    // 1) Filter by salary range (inclusive)
    public List<Employee> filterBySalaryRange(double min, double
max) {
        return employees.stream()
                .filter(e -> e.getSalary() >= min &&
e.getSalary() <= max)
                .collect(Collectors.toList());
    }

    // 2) Filter by department (case-insensitive)
    public List<Employee> filterByDepartment(String dept) {
        String d = dept == null ? "" : dept.toLowerCase();
        return employees.stream()
                .filter(e ->
e.getDepartment().toLowerCase().equals(d))
                .collect(Collectors.toList());
    }
```

```
    // 3) Search by name contains (case-insensitive)
    public List<Employee> searchByName(String q) {
        String qq = q == null ? "" : q.toLowerCase();
        return employees.stream()
                .filter(e ->
e.getName().toLowerCase().contains(qq))
                .collect(Collectors.toList());
    }


    // 4) Compose predicates: filter by dept AND min salary
    public List<Employee> filterByDeptAndMinSalary(String dept,
double minSalary) {
        Predicate<Employee> byDept = e ->
e.getDepartment().equalsIgnoreCase(dept);
        Predicate<Employee> byMinSalary = e -> e.getSalary() >=
minSalary;
        return employees.stream()
                .filter(byDept.and(byMinSalary))
                .collect(Collectors.toList());
    }

    // 5) Count by department
    public Map<String, Long> countByDepartment() {
        return employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.counting()));
    }

}
```

Step 4 — `MainApp.java` (demo)

```
import java.time.LocalDate;
import java.util.List;
import java.util.Map;

public class MainApp {
    public static void main(String[] args) {
        EmployeeService svc = new EmployeeService();

        svc.add(new Employee(1, "Asha", "Engineering", 120000.0,
LocalDate.of(2020, 1, 5)));
        svc.add(new Employee(2, "Vikram", "HR", 70000.0,
LocalDate.of(2019, 3, 12)));
        svc.add(new Employee(3, "Sunita", "Engineering", 95000.0,
LocalDate.of(2021, 6, 1)));
        svc.add(new Employee(4, "Ramesh", "Sales", 65000.0,
LocalDate.of(2018, 11, 20)));
        svc.add(new Employee(5, "Neha", "Engineering", 130000.0,
```

```
LocalDate.of(2017, 8, 15)));
        svc.add(new Employee(6, "Karan", "Sales", 80000.0,
LocalDate.of(2022, 2, 28)));

        System.out.println("=== Employees in Engineering (>=
100k) ===");
        List<Employee> engHigh =
svc.filterByDeptAndMinSalary("Engineering", 100000.0);
        engHigh.forEach(System.out::println);

        System.out.println("\n=== Employees with name containing
'ne' ===");
        svc.searchByName("ne").forEach(System.out::println);

        // === Filter by Salary Range ===
        System.out.println("=== Employees with salary between 70k
and 1L ===");
        List<Employee> salaryRange =
svc.filterBySalaryRange(70000.0, 100000.0);
        salaryRange.forEach(System.out::println);

        // === Filter by Department ===
        System.out.println("\n=== Employees in Engineering
department ===");
        List<Employee> engEmployees =
svc.filterByDepartment("Engineering");
        engEmployees.forEach(System.out::println);

        // === Count by Department ===
        System.out.println("\n=== Count of employees by
department ===");
        Map<String, Long> count = svc.countByDepartment();
        count.forEach((dept, c) -> System.out.println(dept + "
" + c));

    }
}
```

Step 5 — Run it

In IntelliJ: right-click `MainApp.java` → Run `MainApp.main()`

Sample output:

```
=== Employees in Engineering (>= 100k) ===
Employee[id=1,name=Asha,dept=Engineering,salary=120000.00,joined=2020-01-
05]
Employee[id=5,name=Neha,dept=Engineering,salary=130000.00,joined=2017-08-
15]
```

```
=== Employees with name containing 'ne' ===

Employee[id=5,name=Neha,dept=Engineering,salary=130000.00,joined=2017-08-
15]

=== Employees with salary between 70k and 1L ===

Employee[id=2,name=Vikram,dept=HR,salary=70000.00,joined=2019-03-12]

Employee[id=3,name=Sunita,dept=Engineering,salary=95000.00,joined=2021-06-
01]

Employee[id=6,name=Karan,dept=Sales,salary=80000.00,joined=2022-02-28]


=== Employees in Engineering department ===

Employee[id=1,name=Asha,dept=Engineering,salary=120000.00,joined=2020-01-
05]

Employee[id=3,name=Sunita,dept=Engineering,salary=95000.00,joined=2021-06-
01]

Employee[id=5,name=Neha,dept=Engineering,salary=130000.00,joined=2017-08-
15]


=== Count of employees by department ===

Engineering -> 3

Sales -> 2

HR -> 1
```

Lab 7: Git Repo Setup

Initialize repo, create branches, push to GitHub.

Objective

Set up Git for a Java project in IntelliJ IDEA CE: initialize a local repo, commit changes, create branches, and push code to GitHub.

Step 1 — Create a Java project in IntelliJ

1. Open **IntelliJ IDEA CE**.
2. Go to **File → New → Project → Java**.
3. Give the project a name, e.g., `BankAccountApp`.
4. Create a simple Java file, e.g., `Main.java`:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello Git + IntelliJ!");
    }
}
```

Step 2 — Enable Git in the project

1. In the menu bar: **VCS → Enable Version Control Integration**.
2. Choose **Git** from the list.
3. IntelliJ creates a hidden `.git` folder → this initializes a Git repo.

Step 3 — Make the first commit

1. Go to **VCS → Commit...** (or press `Ctrl+K`).
2. Select your files (e.g., `Main.java`, `.idea/` excluded).
3. Write commit message: `Initial commit: setup Java project`
4. Click **Commit** (or **Commit and Push** if remote is already added).

Step 4 — Set up `.gitignore` for Java

Create a `.gitignore` file in the project root:

```
# IntelliJ
.idea/
*.iml

# Java build
out/
target/
*.class
```

Add and commit it:

- Right-click → **Git** → **Add**.
- Then commit with message: `chore: add .gitignore`

Step 5 — Create and switch branches

In IntelliJ:

1. Click the branch name (default: `master` or `main`).
2. Select **New Branch...** → Name: `feature/add-account`.
3. Check "Checkout branch" → IntelliJ switches to the new branch.

Alternatively via terminal (inside IntelliJ terminal tab):

```
git switch -c feature/add-account
```

Step 6 — Add a GitHub remote

Option A: via IntelliJ UI

1. Go to **VCS** → **Git** → **Manage Remotes...**.
2. Add your GitHub repo URL (HTTPS or SSH).
    - Example HTTPS: `https://github.com/<username>/BankAccountApp.git`

Option B: via terminal

```
git remote add origin
https://github.com/<username>/BankAccountApp.git
```

Step 7 — Push code to GitHub

1. In IntelliJ, go to **Git** → **Push** (`Ctrl+Shift+K`).
2. Select branch → Click **Push**.
3. For the first push, IntelliJ will ask to **define upstream** → confirm.

Example via terminal:

```
git push -u origin main
git push -u origin feature/add-account
```

Step 8 — Sync with merged main branch

In IntelliJ:

- Switch back to **main** (`bottom-right branch menu → main`).
- Click **Git** → **Pull** to fetch latest merged changes.

Or via terminal:

```
git switch main
git pull origin main
```

Step 9 — Clean up branches

After merge:

- Delete remote branch:
- `git push origin --delete feature/add-account`
- Delete local branch inside IntelliJ:
  Git → Branches → feature/add-account → Delete

Lab 8: Maven Project Creation

Create & run simple HelloWorld app.

Objective

Learn how to create, configure, build, and run a **Maven Java project** in **IntelliJ IDEA CE** with a simple HelloWorld program.

Step 1 — Create a Maven project

1. Open **IntelliJ IDEA CE**.
2. Go to **File → New → Project**.
3. Select **Maven** on the build system.
4. Fill project details:
    - **GroupId**: `com.example`
    - **ArtifactId**: `hello-maven`
    - **Version**: `1.0-SNAPSHOT`
5. Click **Finish**. IntelliJ creates the project with a `pom.xml`.

Step 2 — Set Java version in `pom.xml`

Open `pom.xml` and add Java version properties:

```
<properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
</properties>
```

Step 3 — Add HelloWorld class

1. Right-click **src/main/java → New → Package →** `com.example`.
2. Inside package → **New → Java Class →** `Main`.
3. Add code:

```
package com.example;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Step 4 — Run HelloWorld (direct run)

- Right-click `App.java` → **Run 'App.main()'**.
- IntelliJ builds and runs the program → output:
- `Hello World!`

Step 5 — Package & run JAR

1. In **Maven Tool Window** → Run **Lifecycle** → **package**.
2. IntelliJ creates a JAR in `target/hello-maven-1.0-SNAPSHOT.jar`.
3. Run from IntelliJ terminal:

```
java -cp target/hello-maven-1.0-SNAPSHOT.jar com.example.Main
```

Lab 9: Convert Maven → Gradle Project

Add Gradle build scripts.

Convert Maven → Gradle Project — Java Lab

Objective

Convert an existing **Maven** Java project into a **Gradle** build by adding Gradle build scripts (Kotlin DSL), creating the Gradle wrapper, importing into **IntelliJ IDEA CE**, and verifying `build`, `test` and `run` work.

Step 1: Automatic conversion (fast)

Use this if you have Gradle installed and want a quick conversion.

From the project root:

```
# (requires Gradle installed)
gradle init --type pom
```

What this does:

- Generates `build.gradle` (or `build.gradle.kts` if you pass `--dsl kotlin`) and `settings.gradle`
- Creates `gradle/` wrapper files **if** supported by your Gradle version
- Tries to translate POM dependencies and basic plugins

After this, go to IntelliJ and import the generated `build.gradle`. Validate and tweak the generated file — automatic conversion is helpful but often needs manual adjustments (plugins, test engine, mainClass, custom plugin behavior).

Step 2: Add the Gradle Wrapper

If `gradle init` did not create a wrapper, create it now (requires Gradle installed). From project root:

```
gradle wrapper
# or specify a version: gradle wrapper --gradle-version 8.5
```

This adds `gradlew`, `gradlew.bat`, and `gradle/wrapper/*` — **commit these files** so everyone uses the same Gradle.

Step 3: Import / refresh in IntelliJ (use the wrapper)

1. Open IntelliJ → **File → Open** and select `build.gradle` (or the project root).
2. Choose **Import project from external model → Gradle**.
3. **Important**: select **Use Gradle wrapper** (recommended).

4. For “Build and run using” and “Run tests using”, choose **Gradle** (keeps behavior consistent with CLI).
5. IntelliJ will import dependencies and create Gradle tool window (you’ll see tasks).

Step 4: Build, test, run (verify)

Use the wrapper from terminal (or IntelliJ Gradle tool window):

```
# run from project root
gradlew clean build       # compile + run tests + produce jar
gradlew jar               # build jar in build/libs/
```

In IntelliJ: use **Gradle Tool Window → Tasks → application → run**, or right-click `App.main()` and run directly.

Lab 10: JUnit Tests for Services

Write unit tests for Calculator or Employee Service class.

Objective

Write **unit tests** (JUnit 5) for small service classes — a pure `Calculator` and an `EmployeeService` — to learn test structure, assertions, lifecycle hooks, and parameterized tests.

Step 1 Add test dependencies in pom.xml

```
<dependencies>
    <!-- JUnit Jupiter -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>

</dependencies>
```

Step 2 Example classes in main/java com.example package

Add Calculator.java

```
package com.example;

public class Calculator {
    public int add(int a, int b) { return a + b; }
    public int subtract(int a, int b) { return a - b; }
    public int multiply(int a, int b) { return a * b; }

    public int divide(int a, int b) {
        if (b == 0) throw new IllegalArgumentException("division
by zero");
        return a / b;
    }
}
```

Add Employee.java

```
package com.example;

import java.time.LocalDate;

public class Employee {
    private final int id;
    private final String name;
    private final String department;
    private double salary;

    public Employee(int id, String name, String department,
```

```
double salary, LocalDate joining) {
        this.id = id; this.name = name; this.department =
department; this.salary = salary;
    }
    public int getId() { return id; }
    public String getName() { return name; }
    public String getDepartment() { return department; }
    public double getSalary() { return salary; }
    public void setSalary(double s) { this.salary = s; }
    public String toString() { return name; }
}
```

EmployeeService.java

```
package com.example;

import java.util.*;
import java.util.stream.Collectors;

public class EmployeeService {
    private final List<Employee> employees = new ArrayList<>();

    public void add(Employee e) { employees.add(e); }
    public List<Employee> getAll() { return new
ArrayList<>(employees); }

    public List<Employee> filterBySalaryRange(double min, double
max) {
        return employees.stream()
                .filter(e -> e.getSalary() >= min &&
e.getSalary() <= max)
                .collect(Collectors.toList());
    }

    public List<Employee> filterByDepartment(String dept) {
        String d = dept == null ? "" : dept.toLowerCase();
        return employees.stream()
                .filter(e ->
e.getDepartment().toLowerCase().equals(d))
                .collect(Collectors.toList());
    }

    public Map<String, Long> countByDepartment() {
        return
employees.stream().collect(Collectors.groupingBy(Employee::getDep
artment, Collectors.counting()));
    }
}
```

Step 3 Tests for `Calculator`

`CalculatorTest.java`

```
package com.example;
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.*;
import org.junit.jupiter.params.provider.CsvSource;

import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
    private Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }

    @Test
    void add_twoPlusThree_returnsFive() {
        assertEquals(5, calc.add(2, 3));
    }

    @Test
    void divide_byZero_throws() {
        IllegalArgumentException ex =
assertThrows(IllegalArgumentException.class, () ->
calc.divide(10, 0));
        assertEquals("division by zero", ex.getMessage());
    }

    @Test
    void multipleAssertions_example() {
        assertAll("basic arithmetic",
                () -> assertEquals(6, calc.multiply(2, 3)),
                () -> assertEquals(1, calc.subtract(3, 2))
        );
    }

    @ParameterizedTest
    @CsvSource({
            "1,2,3",
            "2,3,5",
            "10,-1,9"
    })
    void add_parameterized(int a, int b, int expected) {
        assertEquals(expected, calc.add(a, b));
    }
}
```

Step 4 Tests for `EmployeeService`

`EmployeeServiceTest.java`

```java
package com.example;

import org.junit.jupiter.api.*;
import java.time.LocalDate;
import java.util.List;
import java.util.Map;

import static org.junit.jupiter.api.Assertions.*;

class EmployeeServiceTest {
    private EmployeeService svc;

    @BeforeEach
    void setUp() {
        svc = new EmployeeService();
        svc.add(new Employee(1, "Asha", "Engineering", 120000.0,
LocalDate.of(2020,1,1)));
        svc.add(new Employee(2, "Vikram", "HR", 70000.0,
LocalDate.of(2019,3,3)));
        svc.add(new Employee(3, "Sunita", "Engineering", 95000.0,
LocalDate.of(2021,6,6)));
    }

    @Test
    void filterBySalaryRange_findsEmployeesInRange() {
        List<Employee> list = svc.filterBySalaryRange(80000.0,
125000.0);
        assertEquals(2, list.size());                // Asha &
Sunita
        assertTrue(list.stream().anyMatch(e ->
e.getName().equals("Asha")));
    }

    @Test
    void filterByDepartment_returnsExactDept() {
        List<Employee> eng =
svc.filterByDepartment("Engineering");
        assertEquals(2, eng.size());
        assertTrue(eng.stream().allMatch(e ->
e.getDepartment().equalsIgnoreCase("Engineering")));
    }

    @Test
    void countByDepartment_returnsCorrectCounts() {
        Map<String, Long> counts = svc.countByDepartment();
        assertEquals(2L, counts.get("Engineering"));
        assertEquals(1L, counts.get("HR"));
    }
}
```

Step 5 Run tests

In IntelliJ IDEA CE

- Right-click test class or method → **Run '...Test'**.
- Use the gutter icons (green triangle) next to test methods/classes.
- View results in the **Run** or **Test** tool window.