1. Corporate Data Audit & Reporting System

Context: You're helping a small company process employee data stored in a list. The HR team needs basic reports from this data—like finding specific employees, standardizing name formats, and calculating totals. You are given the Employee class and some sample data.

Starter Code

File: `Employee.java`

```java
package com.company.audit;


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

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

    // Getters (needed for compliance/reporting)
    public String getName() { return name; }
    public int getAge() { return age; }
    public double getSalary() { return salary; }
    public String getDepartment() { return department; }

    @Override
    public String toString() {
        return String.format("Employee{Name='%s', Dept='%s', Salary=$%.2f,
Age=%d}", name, department, salary, age);
    }

    // Test data method
    public static List<Employee> getSampleData() {
        return List.of(
```

```
            new Employee("Aploylice Johnson", 30, 85000.0, "ENGINEERING"),
            new Employee("Bob Smith", 45, 62000.0, "HR"),
            new Employee("Charlie Brown", 22, 91000.0, "ENGINEERING"),
            new Employee("David Lee", 58, 75000.0, "HR"),
            new Employee("Eve Wilson", 33, 55000.0, "SALES"),
            new Employee("
import java.util.List;Frank Miller", 55, 120000.0, "ENGINEERING")
        );
    }
}
```

Questions (Employee Data Processing):

1. **High-Earning Engineers List:**
    - **Goal:** HR needs a list of a ll engineers earning more than $70,000 for review.
    - **Task:** Define a method that accepts a list of employees and a selection criterion (a functional interface) as input. Call this method to retrieve and return a List<Employee> that meets the combined filter: salary > $70,000 AND department = "ENGINEERING".
    - **Constraint:** The filtering logic must be defined inline when calling the method.
2. **Standardized Name Report:**
    - **Goal:** Create a simple report of all employee names, standardized for internal systems.
    - **Task:** Generate a new List<String> containing the full names of every employee. Use a stream operation to ensure every name is converted to UPPERCASE.
3. **Total Annual Salary Budget:**
    - **Goal:** Determine the total annual salary liability for the company's budget review.
    - **Task:** Calculate the company's total annual salary liability by summing the salaries of all employees in the list. Your solution must use stream operations to give a single summary result.

2. Warehouse Inventory Search and Tracking

Context: Your task is to build a high-performance inventory retrieval system. Inventory data is nested and a database lookup is an expensive operation that sometimes fails. Handling deep data structures and potential search failure without resource waste is critical.

Starter Code

File: `Inventory.java`

```java
package com.company.warehouse;

import java.util.List;

public class Inventory {
    private String name;
    private List<List<String>> palletItemIds; // Nested structure: List of
Pallets, where each Pallet is a List of Item IDs

    public Inventory(String name, List<List<String>> palletItemIds) {
        this.name = name;
        this.palletItemIds = palletItemIds;
    }

    public List<List<String>> getPalletItemIds() { return palletItemIds; }

    // Simulates an expensive database lookup
    public static Inventory findItem(String id) {
        if (id.equals("A100")) {
            return new Inventory("Main Inventory", List.of(
                List.of("P10", "P20"),
                List.of("P30", "P10", "P40")
            ));
        }
        return null; // Search fails
    }
}

// File: ItemPlaceholder.java
class ItemPlaceholder {

    // This is a highly resource-intensive object to create.
    public ItemPlaceholder() {
        System.out.println("ALERT: Creating expensive placeholder object!");
    }
    public String getInfo() { return "ID-NOT-FOUND: Placeholder Item"; }
}
```

Questions (Inventory Search & Processing):

1. **Safe Lookup with Default Value:**

    - **Goal:** Implement a search that always returns a valid object, even if the database lookup fails.
    - **Task:** Use the Inventory.findItem(id) method. Wrap the result in a type that explicitly handles the possibility of absence. If the search returns null, you must return a new ItemPlaceholder instance instead.
    - **Constraint:** The expensive ItemPlaceholder object must only be constructed if the search actually failed (lazy computation).
2. **Flattening Inventory IDs:**
    - **Goal:** Extract a clean list of every unique item ID regardless of which pallet it is on.
    - **Task:** Given an Inventory object, process its deeply nested item IDs. Write a sequence of stream operations that extracts a single, flat, non-repeating Set of all unique item IDs across all pallets.

3. Configuration Constraint System (Integrated)

Context: To enforce system integrity, configuration fields must be validated at runtime against strict constraints defined directly within the configuration class structure. You must build the metadata structure and the validation engine.

Starter Code (SystemConfig.java): (Create `RangeCheck.java` and `ConfigValidationException.java` as instructed below)

```java
package com.company.config;

// Assume RangeCheck is defined correctly elsewhere
public class SystemConfig {

    //@RangeCheck annotation must be applied here
    private int maxThreads = 8;

    //@RangeCheck annotation must be applied here
    private int timeoutSeconds = 2500;

    public SystemConfig(int maxThreads, int timeoutSeconds) {
        this.maxThreads = maxThreads;
        this.timeoutSeconds = timeoutSeconds;
    }

    // A simple method for logging successful checks
    public static void logSuccess(String message) {
        System.out.println("SUCCESS: " + message);
```

```
    }
}

// File: ConfigValidationException.java (Unchecked Exception)
// Student must create this simple custom unchecked exception class
// ...
```

Questions (Annotations and Validation):

1. **Defining the Constraint Metadata:**
    - **Goal:** Create the structure to hold minimum and maximum allowed values.
    - **Task:** Define a custom annotation named @RangeCheck.
        - Configure it to be readable by an external tool during runtime.
        - It must only be applicable to class fields.
        - It should hold the integer constraints: min() and max().
2. **Applying the Constraints:**
    - **Goal:** Attach the validation rules to the configuration fields.
    - **Task:** In SystemConfig.java, apply the @RangeCheck annotation to:
        - maxThreads: min 1, max 16
        - timeoutSeconds: min 100, max 5000