## Assignment B: Personal Contact Management System

OOP, Multithreading, and Logging

## Problem Description

Develop a console-based personal contact management system that allows users to add, remove, search, group, and persist contacts.
This assignment focuses on **object-oriented programming**, **concurrent execution using multithreading**, and implementing a **custom logging mechanism using closures and decorators** [**Bonus:** you may extend the logging system to run in a separate logging **thread** using **Python's logging** module]

**The Learning Objectives in this assignment are:**

- Use **data collections** (list, dict, ...)
- Apply **advanced object-oriented programming (**abstraction, inheritance, polymorphism, composition)
- Override special methods (__str__)
- Define and import **custom modules**
- Use **regular expressions** and datetime
- Persist data using **JSON**, Handle **files and exceptions**
- Apply **multithreading using the threading module**
- Implement **custom logging using closures and decorators**
  - [**Bonus**] Apply **thread-based logging**

---

## System Requirements

**0. Comments**
Remember to add descriptive comments to your code that shows the design decisions you make.

---

**1. Data Model (OOP)**
**Abstract Base Class: Contact**
Attributes:

- name
- phone
- email
- created_at (datetime)

Methods:

- __init__
- __str__ (can be overridden)
- get_contact_type() *(abstract)*
- to_dict() *(abstract, for JSON persistence)*

---

**2. Inheritance & Polymorphism**
Implement **at least three contact types** as subclasses of Contact:
**a) PersonalContact**
Additional Attributes:

- birthday (datetime)

**b) WorkContact**
Additional Attributes:

- company
- job_title

**c) EmergencyContact**
Additional Attributes:

- priority_level (integer)

Requirements:

- Each subclass must:
  - Implement get_contact_type()
  - Override __str__
  - Extend to_dict()
- All contact types must be handled **polymorphically** in the system.

---

**3. Composition**
**ContactManager Class**

- Maintains a **list of Contact objects**
- Methods:
  - add_contact(contact)
  - remove_contact(name)
  - search_contacts(keyword)
  - list_contacts()
  - group_by_type()

**4. Regular Expressions**
Validate: Invalid inputs must be handled using **exceptions**.

- Email addresses
- Phone numbers (digits, optional country code)

- Names (alphabetic characters and spaces only)

---

**5. Datetime Handling**

- Store the contact creation timestamp (created_at)
- Parse birthdays from user input
- Sort contacts by creation date
- Optional: filter contacts by a date range

---

**6. File Handling & JSON Persistence**

- Store all contacts in contacts.json
- Implement:
  - load_contacts()
  - save_contacts()
- When loading, the system must **reconstruct the correct subclass type**
- Handle – Exceptions:
  - Missing file
  - Corrupted JSON
  - Unknown contact type
  - File permission errors

---

**7. Multithreading (Core Requirement)**

**Threaded Operations**

**a) Asynchronous Save**

- Saving contacts to contacts.json must occur in a **background thread**
- The main program must remain responsive while saving

Example behaviour:
Saving contacts in background...
You may continue using the system.

**Thread Safety**

- Use threading.Lock to protect:
  - The shared contacts list
  - File write operations
- Ensure consistent data during concurrent access

---

**8. Logging Functionality (Core Requirement)**

**Logging Using Closures & Decorators (Do NOT use Python's logging module here)**

**Closure-Based Logger**

Students must implement a logger using a **closure** that:

- Opens a log file (contacts.log)
- Appends timestamped messages
- Keeps the file handle private

Example conceptual behavior:

```
[2026-01-29 10:32] add_contact called ...
```

**Decorator-Based Logging**

- Implement a decorator that logs:
  - Function name
  - Success or failure
  - Exceptions raised

Apply the decorator to the methods:

- add_contact(contact)
- remove_contact(name)
- add methods that you think are worth for logging

Do **not** print log messages to the console instead of logging them

[**Bonus**] **Thread-Based Logging**

- Use queue.Queue to store log messages
- Implement a **dedicated** logging thread
- The logging thread:
  - Reads messages from the queue
  - Writes them using **Python's logging** module
  - Terminates cleanly when the program exits

---

**9. Custom Modules**

This is only **Suggested** structure (you may adapt if justified):

```
contacts/
 ├── base.py      # Contact (abstract class)
 ├── personal.py  # PersonalContact
 ├── work.py      # WorkContact
 ├── emergency.py # EmergencyContact
 ├── manager.py   # ContactManager
 ├── logger.py    # logging configuration
 ├── utils.py     # regex & datetime helpers
 └── main.py
```

---

**10. Error Handling**

Use try/except to:

- Catch invalid input
- Handle threading-related errors
- Handle file and JSON errors
- Log exceptions with meaningful messages
- Prevent application crashes

---

**Example Menu – when running the program (in console)**

1. Add contact
2. List contacts
3. Search contact
4. Save contacts
5. Exit

---

**Notes:**

- Use multithreading **only where required**
- Avoid unnecessary complexity
- Focus on **correctness, clarity, and safe concurrency**
- Logging quality matters more than quantity
- Use concepts learned in the course.
- Bonus features must be clearly highlighted in your report

## Submission Guidelines:

Via Teams Assignment – Read the below **carefully**:

- Groups of (**max 5 students**) are allowed. One submission per group is enough.
- Submission:
    - A zip file includes:
      **source .py file(s)** of your program (Error free and executable), in addition to any needed *resource*(s), i.e. **everything required to run your solution**.
    - A word file includes:
      - the list of Names and M_Nr. of group's members,
      - a short describing the solution and the **rationale** (**design decisions**) behind the used approaches. [ short bullet points could be useful + highlight the Bonus]
      - **instructions**, including "*how to*" run and test your program.
      In addition to sample **snapshots** of running and testing your program (**tested use cases**).

      Remember the file naming convention (following the communication policy style) to **be the same for both files**:
      **BINT25-F3-Prog-Lib**-LName1-LName2.xxx
      P.S. for larger groups you may use only the initials of the members' names 😊