DAY-23 MORNING ASSESSMENT

21)   
import threading, time  
class LibraryPortal:  
   def \_\_init\_\_(self):  
       self.books = {}  
       self.members = {}  
       self.\_lock = threading.RLock()  
       self.\_load()  
  
   # wrap state‑changing ops  
   def borrow(self, member\_id: str, book\_id: str):  
       with self.\_lock:  
           m = self.members.get(member\_id)  
           b = self.books.get(book\_id)  
           if not m or not b:  
               print("Invalid member/book"); return  
           if not b.available:  
               print("Book not available"); return  
           # (limit check in #23)  
           issue = datetime.now()  
           due = issue + timedelta(days=m.loan\_days())  
           m.\_add\_loan(book\_id, issue, due)  
           b.available = False  
           b.last\_modified = datetime.now().isoformat()  
           self.\_save()  
           self.\_log(f"BORROW {member\_id} -> {book\_id} due={due.date()}")  
           print(f"Issued '{b.title}' to {m.name}. Due: {due.date()}")  
  
   def receive(self, member\_id: str, book\_id: str):  
       with self.\_lock:  
           # ...existing logic...  
           # after computing fine:  
           b.available = True  
           b.last\_modified = datetime.now().isoformat()  
           self.\_save()  
           self.\_log(f"RETURN {member\_id} -> {book\_id} fine={fine}")

Concurrency test (simulated):  
def simulate\_concurrent\_borrows(portal: LibraryPortal, member\_id: str, book\_id: str):  
   def attempt(n):  
       portal.borrow(member\_id, book\_id)  
   threads = [threading.Thread(target=attempt, args=(i,)) for i in range(5)]  
   [t.start() for t in threads]  
   [t.join() for t in threads]  
  
22) import re  
  
def \_valid\_member\_id(mid: str) -> bool:  
   return bool(re.fullmatch(r"[A-Za-z0-9]+", mid))  
  
class LibraryPortal:  
   # ...  
   def insert\_book(self, book\_id, title, author, isbn):  
       with self.\_lock:  
           if book\_id in self.books:  
               print("Book ID exists"); return self  
           # optional: validate isbn with your regex fn from #8  
           self.books[book\_id] = Book(book\_id, title, author, isbn, True)  
           self.books[book\_id].last\_modified = datetime.now().isoformat()  
           self.\_save(); self.\_log(f"ADD\_BOOK {book\_id}")  
           print("Book inserted"); return self  
  
   def enroll\_member(self, mtype: str, member\_id: str, name: str):  
       with self.\_lock:  
           if not \_valid\_member\_id(member\_id):  
               print("Member ID must be alphanumeric"); return self  
           if member\_id in self.members:  
               print("Member ID exists"); return self  
           self.members[member\_id] = MemberFactory.create\_member(mtype, member\_id, name)  
           self.\_save(); self.\_log(f"ENROLL {mtype} {member\_id}")  
           print("Member enrolled"); return self

23)   
class BorrowLimitExceededError(Exception): pass  
  
class LibraryPortal:  
   def borrow(self, member\_id: str, book\_id: str):  
       with self.\_lock:  
           m = self.members.get(member\_id); b = self.books.get(book\_id)  
           if not m or not b: print("Invalid member/book"); return  
           if len(m.borrowed) >= 5:  
               raise BorrowLimitExceededError("Cannot borrow more than 5 books")  
           if not b.available:  
               print("Book not available"); return  
           # proceed as in 21  
  
24) Retry logic if file is locked when saving  
  
import errno, os, time  
  
MAX\_RETRIES = 5  
BACKOFF\_SEC = 0.2  
  
def \_safe\_write(path: str, data: str):  
   for i in range(MAX\_RETRIES):  
       try:  
           with open(path, "w", encoding="utf-8") as f:  
               f.write(data)  
           return  
       except OSError as e:  
           if e.errno in (errno.EACCES, errno.EBUSY):  
               time.sleep(BACKOFF\_SEC \* (2 \*\* i))  
           else:  
               raise  
   raise OSError("Failed to write after retries")  
  
class LibraryPortal:  
   def \_save(self):  
       # build file contents (txt persistence)  
       books\_lines = "".join(b.to\_line() for b in self.books.values())  
       members\_lines = "".join(m.to\_line() for m in self.members.values())  
       try:  
           \_safe\_write(BOOKS\_FILE\_NG, books\_lines)  
           \_safe\_write(MEMBERS\_FILE\_NG, members\_lines)  
       except Exception as e:  
           self.\_log(f"ERROR\_SAVE {e}")  
           raise

25. import json, pathlib  
  
STATE\_JSON = "ng\_state.json"  
BACKUP\_DIR = pathlib.Path("backups")  
BACKUP\_DIR.mkdir(exist\_ok=True)  
  
class LibraryPortal:  
   def \_save\_json(self):  
       payload = {  
           "version": 1,  
           "saved\_at": datetime.now().isoformat(),  
           "books": {bid: vars(b) for bid, b in self.books.items()},  
           "members": {mid: m.to\_dict() for mid, m in self.members.items()},  
       }  
       data = json.dumps(payload, ensure\_ascii=False, indent=2)  
       \_safe\_write(STATE\_JSON, data)  
       ts = datetime.now().strftime("%Y%m%d\_%H%M%S")  
       \_safe\_write(str(BACKUP\_DIR / f"ng\_state\_{ts}.json"), data)  
       self.\_log("SAVE\_JSON version=1")

26. ACTION\_LOG = "ng\_actions.log"  
  
class LibraryPortal:  
   def \_log(self, message: str):  
       ts = datetime.now().isoformat()  
       try:  
           with open(ACTION\_LOG, "a", encoding="utf-8") as f:  
               f.write(f"{ts} {message}\n")  
       except Exception:  
           pass  # avoid crashing on logging errors

27. import csv, json  
  
class LibraryPortal:  
   # CSV export (books)  
   def export\_csv(self, path="library\_books.csv"):  
       with open(path, "w", newline="", encoding="utf-8") as f:  
           w = csv.writer(f)  
           w.writerow(["book\_id","title","author","isbn","available","last\_modified"])  
           for b in self.books.values():  
               w.writerow([b.book\_id,b.title,b.author,b.isbn,b.available,b.last\_modified])  
       self.\_log(f"EXPORT\_CSV {path}")  
  
   # CSV import (books only example)  
   def import\_csv(self, path="library\_books.csv"):  
       with open(path, "r", encoding="utf-8") as f:  
           r = csv.DictReader(f)  
           for row in r:  
               self.books[row["book\_id"]] = Book(  
                   row["book\_id"], row["title"], row["author"], row["isbn"],  
                   row["available"].lower() == "true"  
               )  
               self.books[row["book\_id"]].last\_modified = row.get("last\_modified") or datetime.now().isoformat()  
       self.\_save(); self.\_log(f"IMPORT\_CSV {path}")  
  
   # JSON export/import (full state)  
   def export\_json(self, path="library\_state.json"):  
       data = {  
           "version": 1,  
           "books": {bid: vars(b) for bid, b in self.books.items()},  
           "members": {mid: m.to\_dict() for mid, m in self.members.items()}  
       }  
       with open(path, "w", encoding="utf-8") as f:  
           json.dump(data, f, ensure\_ascii=False, indent=2)  
       self.\_log(f"EXPORT\_JSON {path}")  
  
   def import\_json(self, path="library\_state.json"):  
       with open(path, "r", encoding="utf-8") as f:  
           data = json.load(f)  
       self.books.clear(); self.members.clear()  
       for bid, bd in data["books"].items():  
           self.books[bid] = Book(\*\*bd)  
       for mid, md in data["members"].items():  
           self.members[mid] = MemberFactory.create\_member(md["type"], md["member\_id"], md["name"])  
           self.members[mid].load\_borrowed(md.get("borrowed", []))  
       self.\_save(); self.\_log(f"IMPORT\_JSON {path}")  
  
28. from dataclasses import dataclass, field  
  
@dataclass  
class Book:  
   book\_id: str  
   title: str  
   author: str  
   isbn: str  
   available: bool = True  
   last\_modified: str = field(default\_factory=lambda: datetime.now().isoformat())  
  
   def to\_line(self):  
       return DELIM.join([self.book\_id, self.title, self.author, self.isbn,  
                          "1" if self.available else "0", self.last\_modified]) + "\n"  
  
   @staticmethod  
   def from\_line(line: str) -> "Book":  
       p = line.strip().split(DELIM)  
       # support both old (5 fields) and new (6 fields) formats  
       if len(p) >= 6:  
           return Book(p[0], p[1], p[2], p[3], p[4] == "1", p[5])  
       return Book(p[0], p[1], p[2], p[3], p[4] == "1")

 29. import pickle  
  
class LibraryPortal:  
   def save\_pickle(self, path="library\_state.pkl"):  
       with open(path, "wb") as f:  
           pickle.dump({"books": self.books, "members": self.members}, f, protocol=pickle.HIGHEST\_PROTOCOL)  
       self.\_log(f"SAVE\_PICKLE {path}")  
  
   def load\_pickle(self, path="library\_state.pkl"):  
       with open(path, "rb") as f:  
           data = pickle.load(f)  
       self.books = data["books"]; self.members = data["members"]  
       self.\_save(); self.\_log(f"LOAD\_PICKLE {path}")

 30. def calculate\_fine(member, due\_date: datetime, returned\_at: datetime) -> int:  
   days\_late = max(0, (returned\_at.date() - due\_date.date()).days)  
   if days\_late == 0:  
       return 0  
   base = member.fine\_per\_day()  
   if days\_late <= 3:  
       rate = base                  # 1x for 1–3 days late  
   elif days\_late <= 10:  
       rate = base \* 2              # 2x for 4–10 days late  
   else:  
       rate = base \* 5              # 5x beyond 10 days  
   return days\_late \* rate  
  
# use inside receive()  
now = datetime.now()  
due = datetime.fromisoformat(loan["due\_date"])  
fine = calculate\_fine(m, due, now)