Requirement Explanation

Project: Mini Banking & Payments System

Coding Test on 22nd Sept 2025 - FIL Fresher Training

Time – 90 min | 11 AM ~ 12:30

Mini Banking & Payments System

Python OOP + Exception Handling Exercise

🎯 Exercise Goal

Build a comprehensive mini banking system demonstrating mastery of Python OOP concepts, robust exception handling, and industry best practices. Create a production-ready system with proper architecture and testing.

Core OOP Concepts

- Encapsulation & Data Hiding
- Inheritance & Method Overriding
- Polymorphism & Abstraction
- Operator Overloading

Advanced Python Features

- Mixins & Multiple Inheritance
- Duck Typing & Protocols
- Class/Static Methods & Properties
- Magic Methods (__add__, __repr__)

Focus: Correctness • Clarity • Idiomatic Python • Best Practices

Mini Banking & Payments System

Python OOP + Exception Handling Exercise

📋 Key Components to Implement

Money & Transaction

Decimal-based currency handling with operator overloading and immutable transaction records

Account Hierarchy

Abstract base class with Savings, Checking account types using inheritance patterns

Bank Orchestrator

Factory patterns, account management, and month-end processing with duck-typed strategies

Custom Exceptions

Specific error handling for banking operations with proper exception chaining

Mixins & Composition

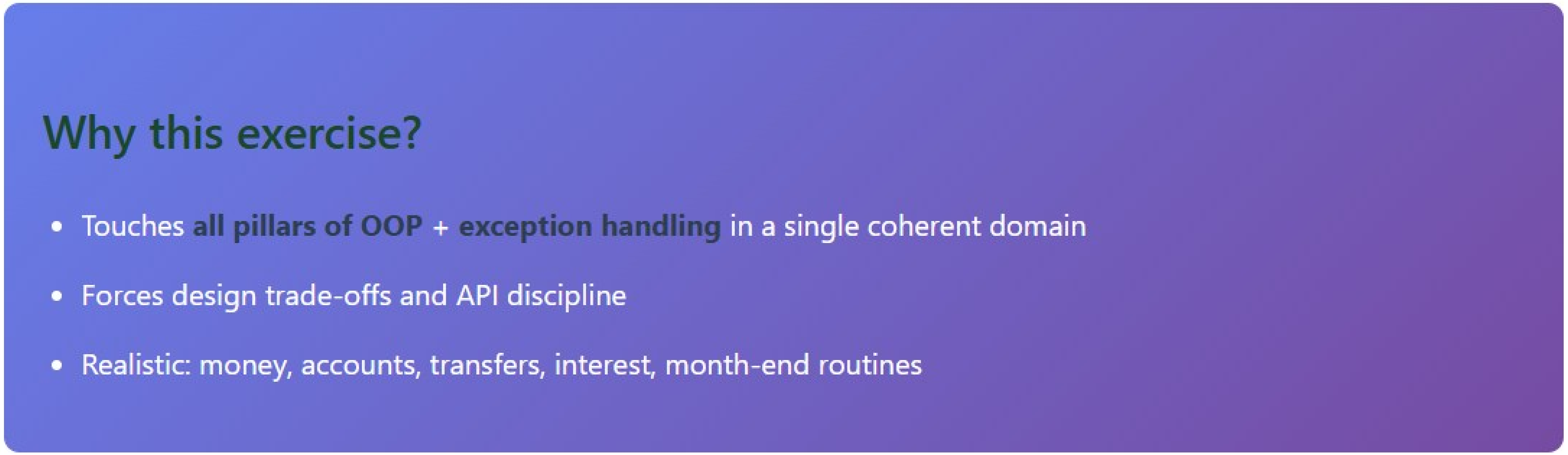
JSONSerializable and Auditable mixins demonstrating multiple inheritance

Unit Testing

Comprehensive test coverage ensuring all functionality works correctly

Focus: Correctness • Clarity • Idiomatic Python • Best Practices

Understanding the Requirement — Mini Banking & Payments System

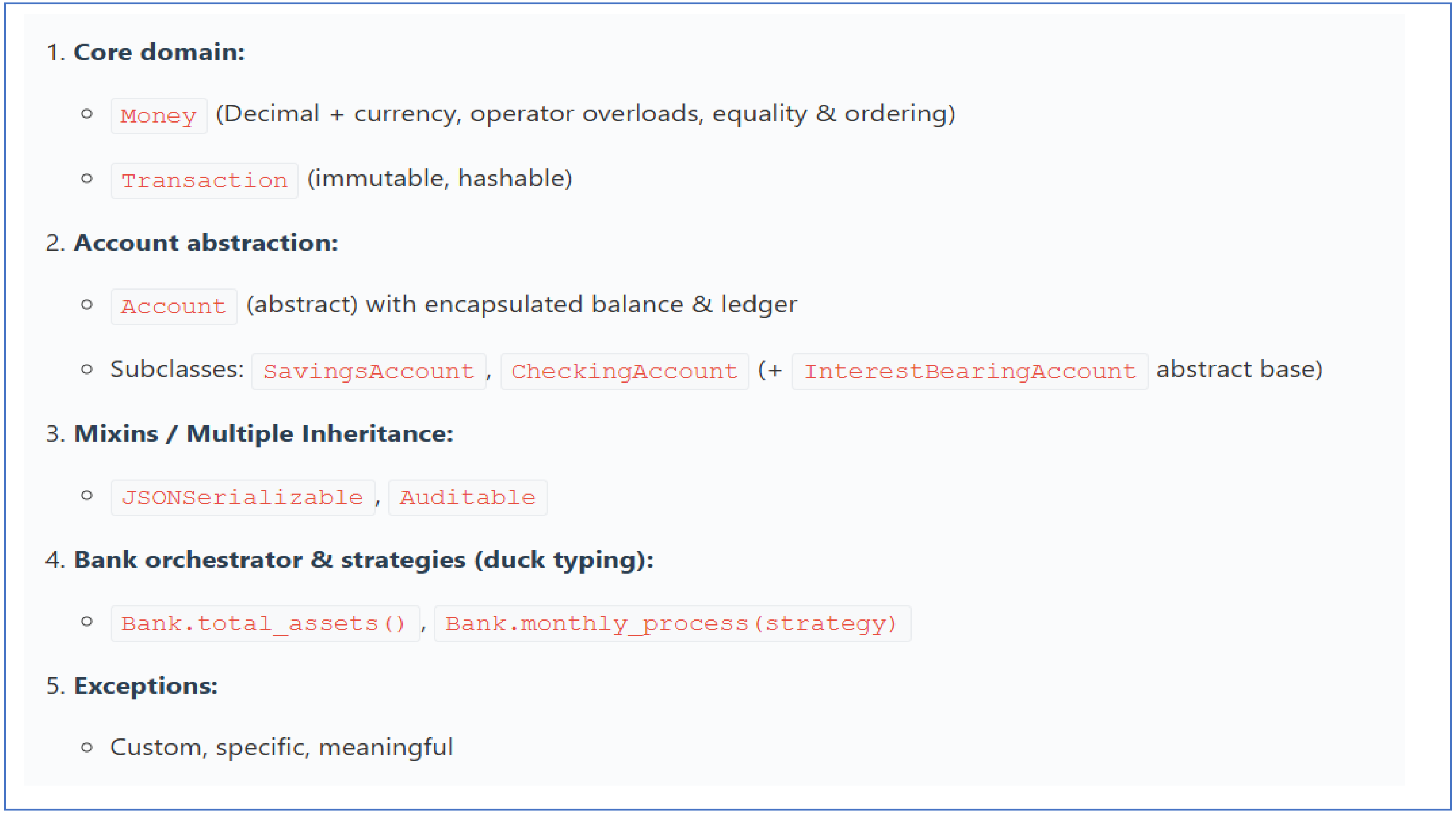


Why this exercise?

- Touches **all pillars of OOP + exception handling** in a single coherent domain
- Forces design trade-offs and API discipline
- Realistic: money, accounts, transfers, interest, month-end routines

Focus: Correctness • Clarity • Idiomatic Python • Best Practices

What you must build



1. **Core domain:**
   - `Money` (Decimal + currency, operator overloads, equality & ordering)
   - `Transaction` (immutable, hashable)
2. **Account abstraction:**
   - `Account` (abstract) with encapsulated balance & ledger
   - Subclasses: `SavingsAccount`, `CheckingAccount` (+ `InterestBearingAccount` abstract base)
3. **Mixins / Multiple Inheritance:**
   - `JSONSerializable`, `Auditable`
4. **Bank orchestrator & strategies (duck typing):**
   - `Bank.total_assets()`, `Bank.monthly_process(strategy)`
5. **Exceptions:**
   - Custom, specific, meaningful

Focus: Correctness • Clarity • Idiomatic Python • Best Practices

Design Constraints

Encapsulation: no public mutation of balances

Architecture: inheritance + composition where it makes sense

Operators: overloading only where it increases clarity

Error Handling: robust; avoid bare `except`

Focus: Correctness • Clarity • Idiomatic Python • Best Practices

Mandatory Dunder Methods

- **Money:** `__add__`, `__sub__`, `__eq__`, ordering, `__repr__`
- **Account:** `__len__`, `__repr__`
- **Transaction:** `__hash__`, `__repr__`

Deliverables to Implement

- Working code with docstrings
- All unit tests must pass
- README describing approach and how to run tests

Focus: Correctness • Clarity • Idiomatic Python • Best Practices

Mandatory Dunder Methods

Hints

- Use `decimal.Decimal` for amounts
- Protect invariants: currencies must match, no negative deposits, overdraft checks
- Use `@property`, `@classmethod`, `@staticmethod` intentionally

Focus: Correctness • Clarity • Idiomatic Python • Best Practices