Week 2 Lab — Object-Oriented Foundations Café POS & Delivery Project

In this lab, you will begin building the foundation of the *Café POS & Delivery System*. The aim is to create a clean domain model using core Object-Oriented Design (OOD) principles such as encapsulation, interfaces, composition, and contracts. This is the very first step in your full semester project where in each week you will incrementally extend and refine this application. Today you are focusing only on the domain core (Money, Product, LineItem, Order), which every later lab will build upon. Getting this right means later labs will feel natural and consistent. By the end, you will have a cohesive, domain-centric POS system that integrates multiple design patterns, refactorings, and architectural concepts taught in lectures. *You may work as individual or a Pair in this Lab. You will submit only one solution if working in Pair.*

Before beginning the exercises of this scaffolding lab, ensure that your development environment is ready. First, verify that Java (preferably 21) is installed by opening a terminal or PowerShell window and running the command java -version. The version should be reported as 21.x. Next, confirm that Maven is available by running mvn -v. If either of these commands fails, correct your installation before proceeding.

Learning Objectives of this Lab

- Apply interfaces and composition to model core business entities.
- Create value objects and enforce contracts (preconditions, invariants).
- Implement a simple domain-driven model (Products, Orders, Line Items, Money).
- Produce an executable demo and unit tests as proof of correctness.

Tasks to Perform in this Lab

1. Define the core domain model: Product, Catalog, LineItem, Order, and Money (Note: Catalog/InMemoryCatalog are already scaffolded for you. They are used by the demo to look up products by id. You do not need to modify them in this lab.)
2. Ensure correctness via contracts (e.g., quantity > 0, price >= 0).
3. Implement a CLI demo that simulates creating an order with products and prints a receipt.
4. Add JUnit tests for Money arithmetic and Order totals.

The purpose of these tasks is not only to practice OOD basics, but to prepare a domain model that will support future labs. For example:

- Next week, the **Strategy pattern** will extend your Order with different payment methods.
- In Week 4, the **Decorator pattern** will wrap Product into richer menu items.
- By Week 10, these same classes will move into the **Domain layer** of a layered architecture.

Keep this bigger picture in mind: today's model is the **core** that survives through the whole semester.

Following are the Step-by-Step Build Instructions

The lab will now guide you through implementing the basic domain model of the Café POS system. Follow each step carefully. The following is an overall structure that may help you in designing the solution for this lab:

```
cafe-pos-week2/
  pom.xml
  src/
    main/
      java/
        com/cafepos/
          common/          ← Value objects (e.g., Money)
          catalog/         ← Product store (e.g., Catalog,
InMemoryCatalog)
          domain/          ← Core entities (Order, LineItem,
OrderIds)
          demo/            ← Week2Demo (console)
    test/
      java/
        com/cafepos/       ← JUnit tests (MoneyTests,
OrderTotalsTests)
```

Once Java and Maven are confirmed, you may work on the Café POS scaffold project in IDE e.g., IntelliJ IDE. When prompted, make sure the Project SDK is set to Java 21. To check that everything builds correctly, run a dry compile using the command mvn -q -DskipTests compile. If you do not see any errors, you are ready to start.

Now the following section will guide you through the detailed steps you need to perform in this lab. Note: the following code (files) are provided to you as a scaffold. Review them carefully, complete missing parts if needed, and ensure they pass tests.

1. **Implement the Money class**
   Begin by defining the `Money` class. Its purpose is to represent monetary values safely. The `Money` class is more than a utility. It is a **value object** that guarantees correctness of all financial calculations in the system. Later when you add discounts, taxes, or loyalty points, everything will flow through Money. That is why we enforce immutability, rounding, and invariants right now.

   – Add methods for `add` and `multiply`.
   – Always use `BigDecimal` internally to avoid floating-point rounding issues.
   – Ensure all amounts are represented with two decimal places.

   - As a rule, do not allow negative amounts. You can confirm your implementation by writing a quick unit test, for example checking that `2.00 + 3.00 = 5.00`.

Money.java
```
import java.math.BigDecimal;
import java.math.RoundingMode;
public final class Money implements Comparable<Money> {
    private final BigDecimal amount;

    public static Money of(double value) { ... }
    public static Money zero() { ... }

    private Money(BigDecimal a) {
      if (a == null) throw new IllegalArgumentException("amount
      required");
      this.amount = a.setScale(2, RoundingMode.HALF_UP);
}
    public Money add(Money other) { ... }
    public Money multiply(int qty) { ... }

    // equals, hashCode, toString, etc.
}
```
Just to inform you in very first lab, the solution for all the example codes provided here (or will be provided in next labs) are tested by placing in a Maven project structure with Java 21 and JUnit 5. This does not stop the code execution in other settings but there is no guarantee that code will execute smoothly on all different Java settings you may have on your systems.

2. **Define Product and SimpleProduct**
   Next, create the `Product` interface and the `SimpleProduct` class that implements it. The `Product` interface is the first example of **extensibility**. Today you only need `SimpleProduct`, but later you will decorate products with add-ons (milk, syrup, size) and create them from recipe codes using the Factory pattern. Designing Product cleanly now avoids headaches later.
   - Each product should have an `id`, a `name`, and a `basePrice`.
   - The constructor should enforce that the `basePrice` is not negative. This gives you the foundation for representing café menu items (you may use the following interface provided as is and work in `SimpleProduct` class constructor).

Product.java
```
public interface Product {
    String id();
    String name();
    Money basePrice();
}
```

SimpleProduct.java

```
public final class SimpleProduct implements Product {
    private final String id;
    private final String name;
    private final Money basePrice;

    public SimpleProduct(String id, String name, Money basePrice)
{ ... }

    @Override public String id() { return id; }
    @Override public String name() { return name; }
    @Override public Money basePrice() { return basePrice; }
}
```

3. **Build the Order class**
   Next, we will build the Order class. An `Order` contains multiple `LineItem`s. `Order` is the central **aggregate** in your domain. Everything else (payment, state transitions, notifications) will attach to `Order`. Keep its methods simple and pure: only totals and item management. This makes later extensions (like payment strategies, undo commands, and order lifecycle states) much easier to add. You need to implement methods for following:
   - `addItem(LineItem)` while checking that the quantity is greater than zero.
   - `subtotal()` which calculates the sum of all line-item totals. See the one line code written using map-reduce reactive style using methods of Money class.
   - `taxAtPercent(int percent)` which applies tax correctly and rounds to two decimals.
   - `totalWithTax(int percent)` which combines the subtotal and tax.
   - The `LineItem` class is provided and can be used as is, you need to work in `Order` class for above instructions

Order.java

```
public final class Order {
    private final long id;
    private final List<LineItem> items = new ArrayList<>();
    public Order(long id) { this.id = id; }
    public void addItem(LineItem li) { ... }
    public Money subtotal() {
      return
      items.stream().map(LineItem::lineTotal).reduce(Money.zero()
      , Money::add);
}
    public Money taxAtPercent(int percent) { ... }
    public Money totalWithTax(int percent) { ... }
}
```

LineItem.java

```
public final class LineItem {
    private final Product product;
    private final int quantity;

    public LineItem(Product product, int quantity) {
        if (product == null) throw new
IllegalArgumentException("product required");
        if (quantity <= 0) throw new
IllegalArgumentException("quantity must be > 0");
        this.product = product; this.quantity = quantity;
    }

    public Product product() { return product; }
    public int quantity() { return quantity; }
    public Money lineTotal() { return
product.basePrice().multiply(quantity); }
}
```

4. **Run the Demo**
   Before running the demo, it's important to understand how products are retrieved. The scaffold provides:
   - **Catalog**: an interface with two simple methods: void add(Product p) and Optional<Product> findById(String id).
   - **InMemoryCatalog**: a concrete implementation backed by a Map<String, Product>.

In the demo, we call catalog.add(...) to load menu items and catalog.findById(...) to retrieve them by id when creating LineItems.

You do **not** need to change these classes in Week 2. Their role is just to separate **lookup and storage concerns** from your domain (Order, LineItem, Product, Money). This keeps the domain focused only on quantities and totals, while product storage and retrieval is delegated to the catalog.

Catalog.java

```
import java.util.Optional;
public interface Catalog {
    void add(Product p);
    Optional<Product> findById(String id);
}
```

InMemoryCatalog.java

```
import java.util.*;
public final class InMemoryCatalog implements Catalog {
    private final Map<String, Product> byId = new HashMap<>();
    @Override public void add(Product p) {
        if (p == null) throw new
IllegalArgumentException("product required");
```

```
        byId.put(p.id(), p);
    }
    @Override public Optional<Product> findById(String id) {
        return Optional.ofNullable(byId.get(id));
    }
}
```

Once you have implemented these classes, run the `Week2Demo` class provided. It should create an order, add products, and print the receipt.

Week2Demo.java

```
public final class Week2Demo {
    public static void main(String[] args) {
        Catalog catalog = new InMemoryCatalog();
        catalog.add(new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50)));
        catalog.add(new SimpleProduct("P-CCK", "Chocolate
Cookie", Money.of(3.50)));
        Order order = new Order(OrderIds.next());
        order.addItem(new LineItem(catalog.findById("P-
ESP").orElseThrow(), 2));
        order.addItem(new LineItem(catalog.findById("P-
CCK").orElseThrow(), 1));
        int taxPct = 10;
        System.out.println("Order #" + order.id());
        System.out.println("Items: " + order.items().size());
        System.out.println("Subtotal: " + order.subtotal());
        System.out.println("Tax (" + taxPct + "%): " +
order.taxAtPercent(taxPct));
        System.out.println("Total: " +
order.totalWithTax(taxPct));
    }
}
```

When executing using CLI, you may open the command prompt in on the path where ".pom" is available, use the following command to compile and run the Lab solution:

Compile: *mvn -q -DskipTests compile*

Run: *java -cp "target/classes" com.cafepos.demo.Week2Demo*

A sample output should look like:

```
Order #1001
Items: 2
Subtotal: 8.50
Tax (10%): 0.85
Total: 9.35
```

Running the demo proves that your domain model works end-to-end. This demo is along with "class diagram" (see deliverables section) is the part of *30-second proof* that lab coordinator will use to check your work in next week.

5. **Add and Run Tests**
   Before finishing, add JUnit tests to confirm that your implementation is correct. For example, test that `Money` addition and multiplication produce the correct results, and that `Order.totalWithTax` produces the expected value. Run the tests in IntelliJ or with the command `mvn -q test`. All tests should pass. A sample test case is shown below:

```
@Test void order_totals() {
    var p1 = new SimpleProduct("A", "A", Money.of(2.50));
    var p2 = new SimpleProduct("B", "B", Money.of(3.50));
    var o = new Order(1);
    o.addItem(new LineItem(p1, 2));
    o.addItem(new LineItem(p2, 1));
    assertEquals(Money.of(8.50), o.subtotal());
    assertEquals(Money.of(0.85), o.taxAtPercent(10));
    assertEquals(Money.of(9.35), o.totalWithTax(10));
}
```

(Pair) Activities and Lab Deliverables

As a result of this lab, you are supposed to complete the following short activity with a partner. Following the UML design guidelines, you need to design the class model that shows classes and their interactions which are used in this Lab. This class diagram you sketch will evolve into layered and pattern-rich diagrams later. Include Catalog (interface) and InMemoryCatalog (implementation) in your diagram as provided components. They will stay unchanged in this lab but make the demo possible. You don't need to use any formal tool for this; you may do it using a paper and pen or any informal tool that you like. In addition to the picture above, you need to provide the short answers to the following questions:

1. Why do we use a dedicated `Money` class instead of storing prices as `double`?
2. What benefits do we get from enforcing constraints such as "quantity must be greater than zero"?
3. Did composition (`Order` has `LineItem`) feel more natural than inheritance? Why or why not?

In addition to the above deliverables, you must submit the following deliverables. All must be presented in one Zip file. Zip file will be named with student(s) ID(s) who worked on the Lab.
1. The complete, compilable codebase with all entities implemented.
2. A demo run showing the printed receipt with correct totals.
3. JUnit test cases passing successfully.