Week 4 Lab — Observer Pattern
Café POS & Delivery Project

In Week 4, you will extend the *Café POS & Delivery System* by introducing the Observer design pattern. The Observer pattern establishes a one-to-many dependency between objects so that when one object (the subject or publisher) changes state, all its dependents (the observers or subscribers) are notified automatically. In our POS system, orders will act as publishers, and components such as the Kitchen Display, Delivery Desk, and Customer Notifier will act as observers that respond to order events.

Before starting, make sure your Week 3 project compiles and runs successfully. You should already have support for multiple payment strategies (Cash, Card, Wallet). Now, open your project in IntelliJ IDEA or any IDE you like and confirm the Project SDK is set to Java 21 (as discussed multiple times we used Java 21 to build the solution). In the terminal, run mvn -q -DskipTests compile to ensure there are no errors. This lab will extend your Order class with **event publishing** capability.

This week you will introduce the **Observer design pattern** into your Café POS system. In a real café, when an order is updated, multiple parts of the system need to know:

– The **Kitchen** must see new items immediately.
– The **Delivery Desk** needs to know when an order is ready.
– The **Customer** should be notified about updates.

Instead of hard-coding all of this logic into the Order class, the Observer pattern allows you to **decouple publishers (orders) from subscribers (kitchen, delivery, customer)**. This way, new observers can be added later without changing the Order logic.

Learning Objectives for this Lab

This lab connects your Week-2 domain and Week-3 payment strategies to a simple eventing mechanism. Orders will act as publishers that announce important changes, while views or services such as the Kitchen Display, Delivery Desk, and Customer Notifier subscribe and react independently. The goal is to keep the Order focused on core domain behavior and let other concerns evolve without modifying Order code. By the end of this lab, you will see how adding a new observer is a low-friction change that does not alter existing domain logic. Following are the specific learning objectives of this lab:

- Understand and apply the Observer design pattern.
- Decouple event publishers from event subscribers.
- Allow multiple observers to respond differently to the same event.
- Preserve the open/closed principle by supporting new observers without modifying existing order logic.

Tasks to Perform in this Lab

In the bigger semester picture, Observer gives you a non-intrusive way to integrate features. Next labs can subscribe new behaviors—such as loyalty notifications or audit logging—without editing the Order class. You are extending the system while keeping its core stable: Order remains the aggregate root for totals and state changes; observers remain replaceable and testable endpoints at the edges.

1. Define the OrderObserver and OrderPublisher interfaces.
2. Modify the Order class to act as a publisher that can register, unregister, and notify observers.
3. Implement concrete observers such as KitchenDisplay, DeliveryDesk, and CustomerNotifier.
4. Implement a CLI demo (Week4Demo) that demonstrates order events triggering notifications to multiple observers.
5. Write JUnit tests using a fake observer to confirm that order events are propagated correctly.

Step-by-Step Instructions to Perform Each Step

1. **Define the Observer Interfaces**

Order is the publisher, and it does not need to know who is listening. The OrderObserver interface is tiny by design so that new observers can be created freely, and the OrderPublisher interface describes how publishers manage subscriptions. Keeping these interfaces small and intention-revealing preserves the open/closed principle and avoids hard-coded dependencies inside Order. To achieve this, code provided below is doing the following for you:

- Create the OrderObserver interface with a method updated(Order order, String eventType).
- Create the OrderPublisher interface with methods to register, unregister, and notifyObservers.

OrderObserver.java

```
public interface OrderObserver {
    void updated(Order order, String eventType);
}
```

OrderPublisher.java

```
public interface OrderPublisher {
    void register(OrderObserver o);
    void unregister(OrderObserver o);
    void notifyObservers(Order order, String eventType);
}
```

2. **Extend the Order Class**

Make the change points explicit. Whenever an event of interest happens—adding an item, paying, or marking the order ready—call notifyObservers with a clear event label. Resist the temptation to put formatting or device-specific logic inside Order; its job is to publish the fact that something happened. Observers will decide how to react. This separation is the essence of decoupling: Order announces; observers interpret. Consider the following specific steps and the snippet provided in this section to complete this task:

- Modify the Order class so it maintains a list of observers.
- Whenever an important event occurs (e.g., item added, paid, marked ready), call notifyObservers.
- Make sure to use **clear event types** like "itemAdded", "paid", and "ready".

Order.java (extension) -- You must complete missing parts using following instructions

```
// 1) Maintain subscriptions
private final List<OrderObserver> observers = new
ArrayList<>();
public void register(OrderObserver o) {
    // TODO: add null check and add the observer
}
public void unregister(OrderObserver o) {
    // TODO: remove the observer if present
}
// 2) Publish events
private void notifyObservers(String eventType) {
    // TODO: iterate observers and call updated(this,
eventType)
}
// 3) Hook notifications into existing behaviors
@Override
public void addItem(LineItem li) {
    // TODO: call super/add to items and then
notifyObservers("itemAdded")
}
@Override
public void pay(PaymentStrategy strategy) {
    // TODO: delegate to strategy as before, then
notifyObservers("paid")
}
public void markReady() {
    // TODO: just publish notifyObservers("ready")
}
```

In this step you are making Order act as a publisher. That means it must keep a list of observers, let them subscribe or unsubscribe, and then broadcast a simple event label whenever something important happens. The event vocabulary in this lab is exactly "itemAdded", "paid", and "ready".

For register, add a null check and then put the observer into the list. Make sure you do not allow null values, and try not to store the same observer twice. For unregister, remove the observer if it exists and do nothing if it does not. This keeps the subscription list clean.

For notifyObservers, loop through the list of observers and call their updated(this, eventType) method. Do not print anything here; the publisher only announces facts, not how they are displayed.

For addItem, call the original add logic and then call notifyObservers("itemAdded"). For pay, delegate to the payment strategy as in Week 3, and afterwards call notifyObservers("paid"). For markReady, simply call notifyObservers("ready"). Always remember: first change the state, then announce the change.

Common mistakes include using == to compare strings instead of .equals, putting print statements inside Order, forgetting null checks, or sending an event before the state has actually changed.

You know you are correct if: registering the same observer twice does not create duplicates, calling addItem notifies observers with "itemAdded", calling pay notifies observers with "paid", and calling markReady notifies observers with "ready".

3. **Implement the Observers**

Each observer exists for a single viewpoint and speaks its own language. The Kitchen Display cares about preparation, the Delivery Desk cares about readiness, and the Customer Notifier cares about polite updates. They are intentionally independent so that any one observer can be added, removed, or replaced without touching the publisher. If you later introduce a push gateway or GUI, it will slot into the same interface. You will update the KitchenDisplay, KitchenDesk, and CustomerNotifier classes (skeleton provided). The classes serves the following purposes:

- **KitchenDisplay**: reacts to "itemAdded" and "paid" events, printing kitchen messages.
- **DeliveryDesk**: reacts to "ready" events, printing delivery preparation messages.
- **CustomerNotifier**: reacts to all events with a polite customer message.

```
// KitchenDisplay.java — You will complete missing parts
public final class KitchenDisplay implements OrderObserver {
    @Override
    public void updated(Order order, String eventType) {
        // TODO: on "itemAdded" -> print "[Kitchen] Order
#<id>: item added"
        //       on "paid"      -> print "[Kitchen] Order
#<id>: payment received"
    }
}
```

```
// DeliveryDesk.java — You will complete missing parts
public final class DeliveryDesk implements OrderObserver {
    @Override
    public void updated(Order order, String eventType) {
        // TODO: on "ready" -> print "[Delivery] Order #<id>
is ready for delivery"
    }
}
// CustomerNotifier.java — You will complete missing parts
public final class CustomerNotifier implements OrderObserver {
    @Override
    public void updated(Order order, String eventType) {
        // TODO: print "[Customer] Dear customer, your Order
#<id> has been updated: <event>."
    }
}
```

In KitchenDisplay, when the event is "itemAdded", print a message that an item was added, and when the event is "paid", print that payment was received. In DeliveryDesk, only react to "ready" and print that the order is ready for delivery. In CustomerNotifier, print a polite customer-facing message for every event type, always including the order id and the event name.
Avoid using == to compare event strings, and do not reach into private fields of Order.
Check yourself: if you add an item, the kitchen and customer should print but delivery should stay silent; if you pay, the kitchen and customer should print but delivery should stay silent; if you mark ready, delivery and customer should print.

4. **Run Week4Demo**

Your demo is the 30-second proof that events flow correctly from Order to multiple listeners. Register the observers, trigger a short sequence—add an item, pay, mark ready—and verify that each observer prints its own appropriate message. The important part is not the text of the message but that a single Order action fans out to several subscribers with no direct coupling. If your implementation is correct, running this demo will show three different perspectives reacting to the same order events, with no if/else logic inside the Order class itself. Week4Demo class is doing the following for you:

- Register all three observers to a new Order.
- Add an item, pay for it, and mark it ready.
- Observe that all three observers are automatically updated with their own messages.

Week4Demo.java

```
public final class Week4Demo {
    public static void main(String[] args) {
        Catalog catalog = new InMemoryCatalog();
        catalog.add(new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50)));

        Order order = new Order(OrderIds.next());
        order.register(new KitchenDisplay());
        order.register(new DeliveryDesk());
        order.register(new CustomerNotifier());

        order.addItem(new LineItem(catalog.findById("P-
ESP").orElseThrow(), 1));
        order.pay(new CashPayment());
        order.markReady();
    }
}
```

You are supposed to work with a partner in this lab Run Week4Demo together and check that all observers print the expected messages.

Demo Expectations

By the end of this lab, you should be able to run Week4Demo. The demo will show how creating an order, adding items, paying, and marking it ready triggers updates across all registered observers. You may use following commands from console to compile/run it:

Compile*: mvn -q -DskipTests compile*
Run*: java -cp target/classes com.cafepos.demo.Week4Demo*
An example output is shown below:

```
[Kitchen] Order #1005: 1x Espresso added
[Customer] Dear customer, your Order #1005 has been updated:
itemAdded.
[Kitchen] Order #1005: Payment received
[Delivery] Order #1005 is ready for delivery
[Customer] Dear customer, your Order #1005 has been updated:
ready.
```

5. Testing

Testing should focus on whether observers receive the correct events, not on console output. Create a fake observer that records the events it receives into a list. Register it on an order, trigger one action, and check that the expected event label is in the list. Write two tests: one that proves a single event like "itemAdded" is delivered, and another that proves multiple observers both receive the "ready" event. This style of test will remain valid even if you later change the exact wording of printed messages, because you are testing event flow rather than message formatting. To confirm correctness, you will write tests using a fake observer based on following steps:

1. Create an observer that simply records received event types in a list.
2. Register this fake observer on an order.
3. Trigger an event (e.g., addItem) and assert that "itemAdded" appears in the list.

4. Run all tests using mvn -q test.

This ensures that notifications are propagated without requiring manual console checks.

A sample test using a fake observer is shown below:

```
@Test void observers_notified_on_item_add() {
    var p = new SimpleProduct("A", "A", Money.of(2));
    var o = new Order(1);
    o.addItem(new LineItem(p, 1)); // baseline

    List<String> events = new ArrayList<>();
    o.register((order, evt) -> events.add(evt));

    o.addItem(new LineItem(p, 1));
    assertTrue(events.contains("itemAdded"));
}
```

Deliverables

By the end of this lab, you must submit:
1. The extended codebase with the observer package and concrete observers.
2. A demo run showing that adding items, paying, and marking ready trigger appropriate notifications.
3. Include an updated class diagram (lab coordinator will check it) showing Order as the publisher, the observer interfaces, and the three concrete observers. Make sure the diagram still shows the Strategy elements from Week 3 and the Catalog→Product link used by the demo. This helps you keep a consistent mental model as the project grows.
4. JUnit tests demonstrating that observers are notified correctly.
5. A short reflection (3–4 lines) on rationalizing following:
   - How does the Observer pattern improve **decoupling** in the Café POS system?
   - Why is it beneficial that new observers can be added without modifying the Order class?
   - Can you think of a real-world system (outside cafés) where Observer is used (e.g., push notifications, GUIs)?