2) Dependency Injection Explanation

Yes, this code demonstrates dependency injection. Here's why:

- The `Customer` class doesn't create its own `Profile` instance but receives it through its constructor
- The `Profile` class doesn't create its own `ShippingInfo` instance but receives it through its constructor
- This is constructor injection, where dependencies are "injected" into a class rather than the class creating them itself
- This makes the code more flexible and testable, as we can easily substitute different implementations of the dependencies

3) Law of Demeter Violation and Fix

The current code violates the Law of Demeter (principle of least knowledge) because:

- In `EComController`, we're chaining method calls: `customer.getProfile().getShippingInfo().getCity()`
- This means `EComController` has knowledge about the internal structure of `Customer`, `Profile`, and `ShippingInfo`
- If any of these internal structures change, `EComController` would need to be modified

A possible fix would be to add a method in the `Customer` class that provides the city directly:

java Copy Download

```java
public class Customer {
    // ... existing code ...

    public String getShippingCity() {
        return profile.getShippingInfo().getCity();
    }
}
```

Then in `EComController`, we can simply call:

java Copy Download

```java
System.out.println("Shipping to: " + customer.getShippingCity());
```

This way:

- `EComController` only needs to know about `Customer`
- The knowledge of how to get the city is encapsulated within `Customer`
- If the internal structure changes, only `Customer` needs to be modified, not `EComController`

Additional Questions

1. What happens if an exception is thrown but not caught?

If an exception is thrown but not caught:

- The exception propagates up the call stack
- If it reaches the top of the call stack (typically the main method) without being caught
- The program terminates and prints the exception stack trace to the standard error stream
- This is generally undesirable as it provides a poor user experience

2. Advantages of catching specific exceptions

Catching specific exceptions provides several advantages:

- More precise error handling - different exceptions can be handled differently
- Better code readability - it's clear what specific errors are being handled
- More maintainable code - new exception types can be added without breaking existing handlers
- Better debugging - the exact type of error is preserved
- Prevents accidentally catching and handling exceptions you didn't intend to handle

3. Try-with-resources usage and advantages

Try-with-resources should be used when working with resources that need to be closed after use (like streams, files, database connections).

Advantages:

- Automatic resource management - resources are closed automatically
- Cleaner code - no need for explicit finally blocks
- Prevents resource leaks
- Multiple resources can be declared in a single try statement
- Resources are closed in the reverse order of their declaration
- Any exceptions during closing are suppressed and added to the primary exception

Aufgabe 3:

Aufgabenteil b) Welche Variante macht hier mehr Sinn?

In diesem Fall (Taschenrechner-Funktion `divide`) ist **Variante 2 (Exception an den Aufrufer delegieren)** die bessere Wahl.

Begründung:

1. **Separation of Concerns (Trennung der Verantwortlichkeiten)**
   - Die `divide`-Methode sollte nur für die Berechnung zuständig sein.
   - Die Fehlerbehandlung (z. B. Benachrichtigung des Nutzers, Logging) gehört in den aufrufenden Code (`main` oder andere Anwendungsschichten).
2. **Flexibilität für den Aufrufer**
   - Eine GUI-Anwendung könnte eine Fehlermeldung anzeigen.
   - Eine Server-Anwendung könnte den Fehler loggen und eine Standardantwort senden.
   - Würde `divide` den Fehler intern behandeln, könnte der Aufrufer nicht mehr entscheiden, wie reagiert wird.
3. **Klarere API-Dokumentation**
   - Durch `throws ArithmeticException` ist direkt erkennbar, dass die Methode fehlschlagen kann.
   - Bei interner Behandlung (Variante 1) müsste man die Methode dokumentieren, z. B. "Gibt `NaN` bei Division durch Null zurück".

➡️ **Fazit:** Da `divide` eine generische mathematische Funktion ist, sollte sie keine Annahmen über die Fehlerbehandlung treffen.

Aufgabenteil c) Beispiel, wo die andere Variante (lokale Behandlung) sinnvoller ist

Ein Szenario, in dem **Variante 1 (Exception innerhalb der Methode behandeln)** besser passt:

Beispiel: Temperatur-Sensor mit Default-Wert

java Copy Download

```java
public class TemperatureSensor {
    /**
     * Liest einen Sensorwert aus. Falls der Sensor defekt ist (Division durch Null simuliert),
     * wird ein Standardwert zurückgegeben.
     */
    public static double readSafeValue() {
        try {
            double rawValue = readFromSensor(); // Kann intern eine Exception werfen
            return rawValue / 0; // Simulierter Fehlerfall
        } catch (ArithmeticException e) {
            return 25.0; // Standard-Raumtemperatur, falls Sensor ausfällt
        }
    }

    private static double readFromSensor() {
        // Simuliert eine fehlerhafte Sensoroperation
        throw new ArithmeticException("Sensor error: division by zero");
    }
}
```

Warum ist hier Variante 1 besser?

1. **Die Methode hat spezifisches Fehlerwissen:**
    - Sie "weiß", dass ein Sensorausfall durch einen Default-Wert ersetzt werden soll.
    - Der Aufrufer muss sich nicht um Fehler kümmern.
2. **Vermeidung von boilerplate-Code im Aufrufer:**
    - Jeder Aufruf von `readSafeValue()` würde sonst ein `try–catch` erfordern.
    - Hier ist die Exception ein **interner Fehler**, der nicht nach außen sichtbar sein muss.

3. **Stabilität im Echtzeit-System:**
    - In einer Steuerungssoftware (z. B. Heizungsregelung) soll ein Sensorausfall nicht zum Absturz führen.

Zusammenfassung der beiden Varianten

Kriterium	Variante 1 (lokale Behandlung)	Variante 2 (Exception weiterwerfen)
Verwendung	Spezifische Fehlerlogik (z. B. Default-Werte)	Generische Funktionen (z. B. Mathe-Bibliothek)
Vorteil	Vereinfacht den Aufrufer-Code	Maximale Flexibilität für den Aufrufer
Nachteil	Aufrufer weiß nicht, dass ein Fehler auftrat	Aufrufer muss Exception handhaben
Beispiel	Temperatur-Sensor mit Fallback-Wert	Taschenrechner-Division