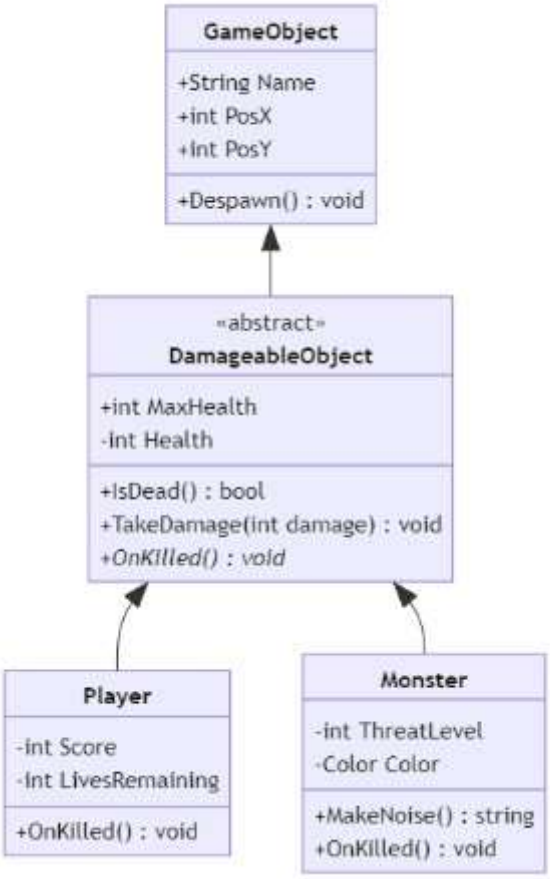
QUIZ QUESTIONS 2
OBJECT-BASED PROGRAMMING PRACTICUM

Dimas Arya Sadewa
SIB 2G
2341760173 - 08

1. Identify the following Abstract method and Class usage, explain the purpose of the diagram class and create the program code to the demo to display it.



Purpose

The code simulates a simple game scenario where a **Player** and a **Monster** interact with each other in a battle setting. Both **Player** and **Monster** can take damage and be "killed" when their health reaches zero. The code also demonstrates inheritance, encapsulation, and polymorphism in object-oriented programming.

1. **GameObject**:
   - **Purpose**: Serves as a base class for any object that can exist in the game world, providing fundamental attributes like name and position (X, Y coordinates).
   - **Key Elements**:
     - Attributes for storing name and position.

    - A despawn method, intended to remove the object from the game (placeholder functionality).
2. **DamageableObject**:
    - **Purpose**: An abstract class that extends GameObject, adding health-related functionality. It represents objects that can take damage and potentially be destroyed.
    - **Key Elements**:
        - Health attributes (maxHealth and health) to keep track of the object's vitality.
        - A method to determine if the object is "dead" based on its health.
        - A takeDamage method to reduce health when the object takes damage, and checks if the object has "died."
        - An abstract onKilled method that must be implemented by subclasses to define what happens when the object is killed.
3. **Player**:
    - **Purpose**: Represents a player in the game, with additional properties such as score and remaining lives.
    - **Key Elements**:
        - Attributes for tracking the player's score and lives remaining.
        - An implementation of the onKilled method to define what happens when the player "dies," such as displaying a game-over message.
4. **Monster**:
    - **Purpose**: Represents a monster enemy in the game, with specific attributes like threat level and color.
    - **Key Elements**:
        - Attributes to store the monster's threat level and color.
        - A makeNoise method to represent a monster-specific sound (e.g., "Roar!").
        - An implementation of the onKilled method to handle what happens when the monster is "killed," like dropping items or displaying a message.
5. **Main**:
    - **Purpose**: The entry point of the program, where instances of Player and Monster are created, and a simple battle simulation is conducted.
    - **Key Elements**:
        - Creates and initializes a player and a monster with names, health values, and other relevant properties.
        - Simulates an exchange of attacks between the player and the monster by calling takeDamage.
        - Checks if either the player or the monster has "died" and prints the outcome, continuing the "battle" if both are still alive.

(Document Code On Github)

2. A client of yours is a Seller who has a lot of media to accommodate orders from customers, but this Seller has difficulty in creating Order categories, he wants every order to have an order date and there must be a confirmation method for each category which is separated into 3 classes: MailOrder, WebOrder, WhatsappOrder. There is an "order status tracking" contract on the MailOrder and WebOrder classes

   Help your client by describing his diagram classes that are easy for him to understand!

   1. **Order (Abstract Class)**:
      - **Purpose**: Serves as the base class for all types of orders. It includes common properties and methods for orders, regardless of the platform.
      - **Attributes**:
        - orderDate - Date when the order was placed.
      - **Methods**:
        - confirmOrder() - Abstract method to confirm the order. This must be implemented in each subclass.
        - cancelOrder() - Cancels the order (optional common behavior across all order types).
   2. **TrackableOrder (Interface)**:
      - **Purpose**: Represents a "tracking" contract. Classes that implement this interface must have tracking functionality, suitable for orders that need status tracking.
      - **Methods**:
        - trackOrderStatus() - Method to check or update the current status of an order.
   3. **MailOrder (Concrete Class)**:
      - **Inherits**: Order (implements TrackableOrder interface)
      - **Purpose**: Represents orders made via mail. Includes tracking because it implements TrackableOrder.
      - **Attributes**: May include mail-specific details if necessary.
      - **Methods**:
        - Implements confirmOrder() - Specific logic for confirming mail orders.
        - Implements trackOrderStatus() - Logic for tracking the mail order status.
   4. **WebOrder (Concrete Class)**:
      - **Inherits**: Order (implements TrackableOrder interface)
      - **Purpose**: Represents orders made through a website. It supports tracking as it implements TrackableOrder.
      - **Attributes**: May include web-specific details such as order ID, payment status, etc.
      - **Methods**:
        - Implements confirmOrder() - Logic to confirm the web order.
        - Implements trackOrderStatus() - Logic to track the web order status.
   5. **WhatsappOrder (Concrete Class)**:
      - **Inherits**: Order

- **Purpose**: Represents orders made through WhatsApp. Unlike MailOrder and WebOrder, this class does not need tracking, as requested by the client.
- **Methods**:
  - Implements confirmOrder() - Specific logic to confirm a WhatsApp order.

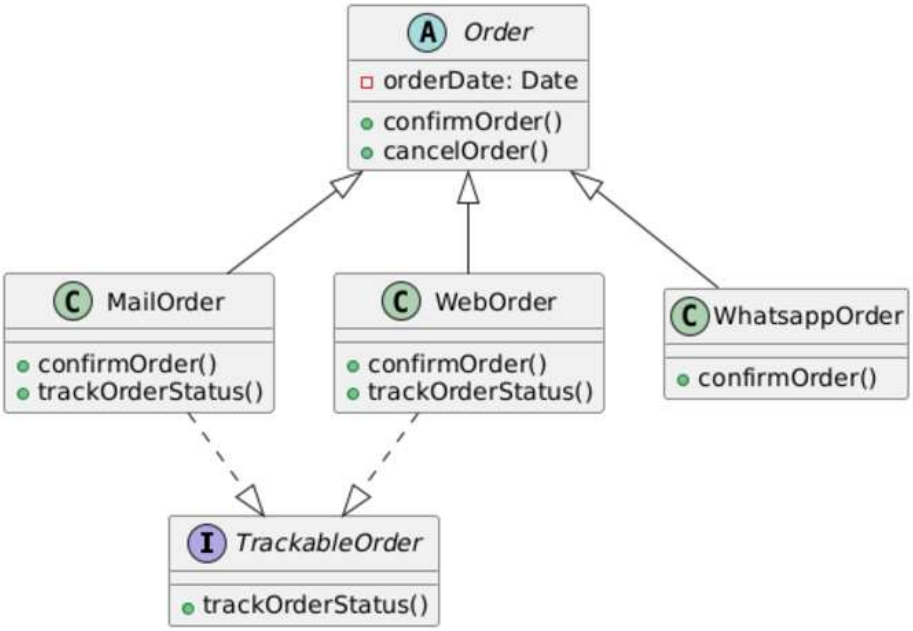
Summary Of The Diagram

- **Order** is an abstract class that holds the common properties and behavior for all orders.
- **TrackableOrder** is an interface that ensures specific orders (MailOrder and WebOrder) can track their status.
- **MailOrder** and **WebOrder** inherit from Order and implement TrackableOrder, making them trackable.
- **WhatsappOrder** inherits from Order but does not implement tracking, following the client's requirement.

This will make it simpler to handle confirmation methods for various order kinds, track statuses, and manage order categories, all of which will aid the customer in understanding the solution.

Explanation for Client

- Easily add common properties for all orders (like orderDate) in one place.
- Create specific order types with different confirmation methods, as each subclass (MailOrder, WebOrder, WhatsappOrder) has its own way of confirming orders.
- Track order status for orders that require it (MailOrder and WebOrder) by using a TrackableOrder interface.



Overview:

The program simulates an order system with three types of orders: **MailOrder**, **WebOrder**, and **WhatsappOrder**. These orders are handled differently, but they all share some common properties and behaviors. The key concept here is **polymorphism** and the use of **abstract classes** and **interfaces** to manage these differences efficiently.

1. Order Class (Abstract Class):

- **Purpose:** This is the base class for all types of orders. It defines the common behaviors (methods) and properties (attributes) that all orders will share.
- **Properties:**
  - Date orderDate: This stores the date when the order was placed.
- **Methods:**
  - confirmOrder(): This is an abstract method that each subclass (like MailOrder, WebOrder, etc.) must implement to confirm the order.
  - cancelOrder(): This method allows cancelling the order. It prints a message indicating the order has been canceled.

2. TrackableOrder Interface:

- **Purpose:** This interface defines a contract for orders that can track their status. It only contains the trackOrderStatus() method.
- **Implemented By:** The MailOrder and WebOrder classes implement this interface, meaning these classes provide a concrete implementation for tracking order status.

3. MailOrder Class:

- **Purpose:** This class represents an order made by mail.
- **Implementation:**
  - It extends the Order class, inheriting its properties and methods.
  - It **implements** the TrackableOrder interface, meaning it must implement the trackOrderStatus() method.
  - The confirmOrder() method is overridden to print a confirmation message when the order is confirmed.
  - The trackOrderStatus() method is overridden to print a message indicating the tracking of a mail order.

4. WebOrder Class:

- **Purpose:** This class represents an order made through the web (online).
- **Implementation:**
  - Like MailOrder, it extends the Order class and implements the TrackableOrder interface.
  - It overrides confirmOrder() to print a confirmation message when the order is confirmed.
  - It also provides an implementation of trackOrderStatus() to simulate tracking of the web order.

5. WhatsappOrder Class:

- **Purpose:** This class represents an order made via WhatsApp.
- **Implementation:**
  - This class extends the Order class, just like MailOrder and WebOrder.

- It does **not** implement the TrackableOrder interface because WhatsApp orders do not require order tracking functionality.
- It overrides the confirmOrder() method to print a confirmation message for WhatsApp orders.

6. Main Class:

- **Purpose:** This is the entry point for the program. It is where the different types of orders are created, confirmed, tracked (if applicable), and canceled.
- **Execution Flow:**
  - Three different orders are created: MailOrder, WebOrder, and WhatsappOrder. Each of them is initialized with the current date using new Date().
  - The confirmOrder() method is called on each order to confirm it, printing a confirmation message.
  - For MailOrder and WebOrder (since they are **TrackableOrder** types), the trackOrderStatus() method is called to track the order's status.
  - Finally, the cancelOrder() method is called on each order to simulate canceling the order.

Key Concepts in Action:

1. **Inheritance:**
   - The MailOrder, WebOrder, and WhatsappOrder classes **inherit** from the Order class, meaning they automatically get the properties (orderDate) and methods (cancelOrder()) from Order.
2. **Polymorphism:**
   - The confirmOrder() method is **overridden** in each subclass to provide specific behavior for each type of order (mail, web, or WhatsApp). This is an example of **method overriding** in Java, where the child classes provide their own implementations of the method defined in the parent class.
   - For the **TrackableOrder** interface, both MailOrder and WebOrder classes implement the trackOrderStatus() method. This allows polymorphic behavior when tracking orders. The appropriate tracking message is printed based on the actual type of the order.
3. **Interfaces:**
   - The TrackableOrder interface defines a common contract for classes that need to implement order tracking. This allows the program to know that objects of MailOrder and WebOrder types can track their status, while objects of WhatsappOrder cannot.
4. **Abstract Classes:**
   - The Order class is an **abstract class**, which means it cannot be instantiated directly. It serves as a template for the subclasses (MailOrder, WebOrder, and WhatsappOrder) to inherit from and implement the required behaviors.
5. **Conclusion:**
   This system is designed to handle different types of orders (Mail, Web, WhatsApp) while providing common behaviors such as confirming, canceling, and tracking order status. By using **inheritance**, **polymorphism**, **abstract classes**,

and **interfaces**, the code is flexible and easy to extend with new types of orders if needed in the future.

(Document Code On Github)

3. Give an example of program code using the concept of polymorphism (Heterogenous Collection, Object Casting, Polymorphic Arguments, InstanceOf) on 1 theme (for example, choose 1 theme: vehicle or electronic device or animal, etc... You can create any theme to apply the 4 points of polymorphism). Create interrelated java program code.

I Will Create a Polymorphism Example based on the theme of "Electronic Devices". In this example, we will use the following concepts:

1. **Heterogeneous Collection: A collection that can store different types of objects.**
2. **Object Casting: Converting between parent class and subclass objects.**
3. **Polymorphic Arguments: Passing objects of different subclasses to the same method.**
4. **instanceof Operator: Checking the type of an object at runtime.**

Theme: Electronic Devices
We'll define a hierarchy of electronic devices like Phone, Laptop, and Tablet, which all inherit from a base class ElectronicDevice. Then, we'll demonstrate polymorphism with heterogeneous collections, object casting, polymorphic method arguments, and instanceof

Explanation of Concepts:

1. **Heterogeneous Collection:**
   - **In the PolymorphismExample class, we created a list called devices that stores various types of ElectronicDevice objects (Phone, Laptop, Tablet). This is an example of a heterogeneous collection, where we can store objects of different subclasses under the same parent class type (ElectronicDevice).**
2. **Object Casting:**
   - **I created an ElectronicDevice reference called myDevice and assigned it a Laptop object. Later, we used instanceof to check if the object is an instance of Laptop, and then cast it to a Laptop type to access the Laptop-specific features (e.g., the brand and model properties).**
3. **Polymorphic Arguments:**

- **The method turnOn() is polymorphic. We call it on all devices (Phone, Laptop, Tablet) stored in the devices list, even though each subclass has its own implementation of turnOn(). This shows how the same method can behave differently depending on the object type (runtime polymorphism).**
- **device.turnOn() is the polymorphic call because it behaves differently depending on whether the device is a Phone, Laptop, or Tablet.**

4. **instanceof Operator:**
    - **I used the instanceof operator to check the type of an object at runtime. In the loop, we check if a device is an instance of Phone, Laptop, or Tablet, and then print out the corresponding type. This ensures type safety when casting objects and handling different subclasses.**

(Document Code On Github)

---- Good Luck ----