6.2P: Key Object-Oriented Concepts

Four Key Principles of OOP:

1. **Encapsulation:**

- Encapsulation is the process of putting data and the methods that utilize such data into a singe class. We can then decide which bit of information from this class is publicly accessible or otherwise hidden. This optimizes the amount of information that is exposed in other classes/files and facilitates flexibility and scalability, which is crucial to consider when designing large programs.
- Example: In the clock program we have created in task 3.1P, we have the Counter class which is initialized with a name and integer. We later created methods which enable the integer to increment and reset.
  ➔ This class later can be initialized in other files and reused countless times to create the clock. However, the core functionalities of the class will not be displayed, only those which are defined as 'public' – The functions are called and used. This makes the program more optimized and modular.

2. **Inheritance:**

- Inheritance is an inherent feature of OOP classes in which they can inherit others' functionality and extend or reform them. This practice promotes high reusability and a hierarchal structure in which classes are based off previous iterations. I love OOP because of this. This makes it easier to track problems within the code because to pinpoint issues present in child classes, we can always trace back to its parent classes. It also makes it easier to improve on the functionalities of a class or completely revamp it for multiple purposes.
- Example: A perfect example of Inheritance is in The Shape Drawer we have done. We made a base class called 'Shape' which later other class shapes (Rectangle, Circle, Line) are based on. Within this class shapes are basic variables and methods that every shape has (color,

coordinates, and Draw, Save, Load method). These variables and methods can later be reused within the children classes.

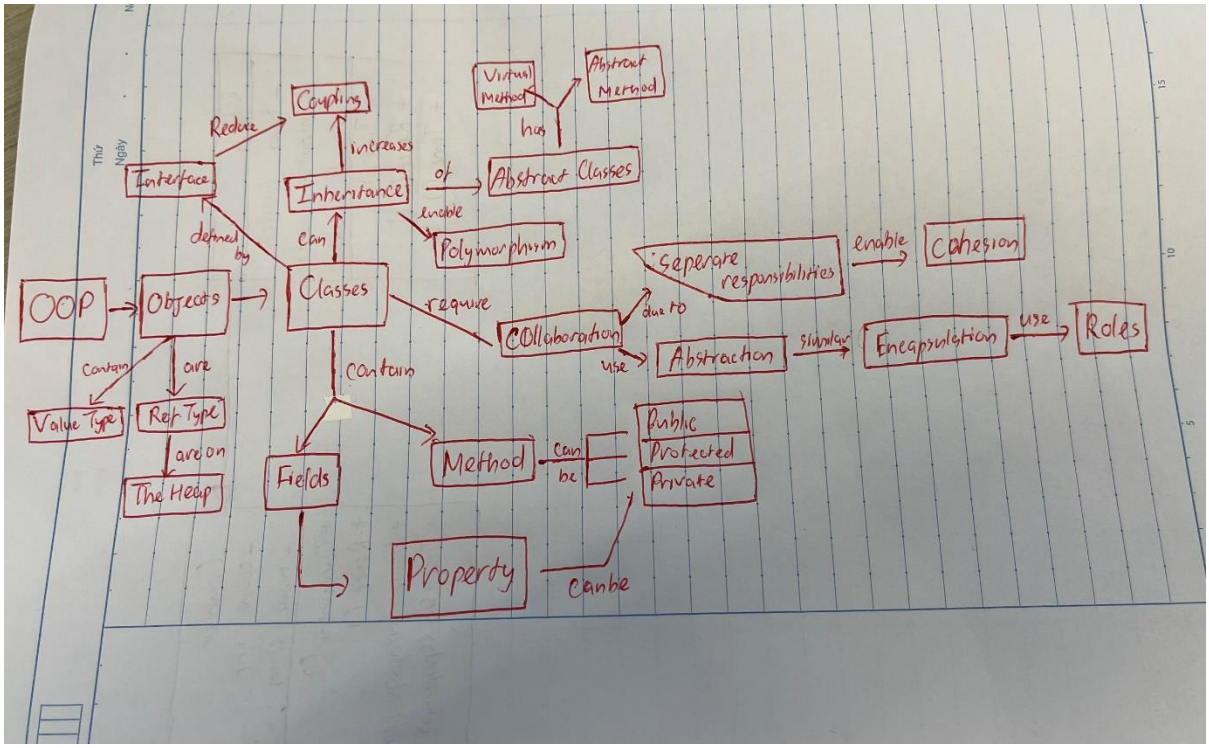
3. **Polymorphism:**

- In my personal view, I think polymorphism is based on Inheritance in a way. It still enables classes to inherit a parent class but allow that class to take on various different forms depending on the context that it is reused in. In other words, while Inheritance relationship is “is-a”, the relationship that Polymorphism is more of “act-like”: classes can perform different actions while still being categorized as the same type.
- Same example as Inheritance but we’re going to talk about the Draw method. While in Shape class it’s declared as Abstract and thus has no code. In Rectangle class, the Draw class is overridden to draw a rectangle using its own width and height variable. Similarly, in the Circle class, the Draw class is reused but to draw a circle with its own radius variable. As you can see, each shape class utilize the Draw method in its own way. When we call them in program file: MyRectangle.Draw() will perform differently to MyCircle.Draw(). This is Polymorphism.

4. **Abstraction:**

- Abstraction describes the state in which a method is declared without any implementation. The method details will be specified later by the other methods which inherit it. Therefore, abstraction allows programmers to focus on what does an object do, instead of how it do it. I think abstraction simplifies the program a lot and facilitates scalability by reducing code complexity.
- The example used for Polymorphism describes Abstraction pretty well. The original Draw method in ‘Shape’ class has no actual implementation but rather acts as a barebone structure which other subclass shape can import their own implementation into.

Diagram:


Virtual Method
Abstract Method
has
Coupling
Reduce
increases
Interface
Inheritance
of
Abstract Classes
enable
Polymorphism
defined by
can
Classes
OOP
Objects
require
Collaboration
due to
Seperate responsibilities
enable
Cohesion
Contain
are
Value Type
Ref Type
contain
use
Abstraction
similar
Encapsulation
use
Roles
are on
The Heap
Fields
Method
can be
Public
Protected
Private
Property
can be