บทที่ 1

หลักการเขียนโปรแกรมเชิงวัตถุ

Principles of Object-Oriented Programming

สอนโดย อ.เรวดี พิพัฒน์สูงเนิน สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ มหาวิทยาลัยราชภัฏอุดรธานี

หลักการเขียนโปรแกรมเชิงวัตถุ

วัตถุประสงค์การเรียนรู้

- เข้าใจแนวคิดพื้นฐานของการเขียนโปรแกรมเชิงวัตถุ (OOP)
- อธิบายข้อดีและข้อเสียของการเขียนโปรแกรมเชิงวัตถุ
- รู้จักและเข้าใจส่วนประกอบพื้นฐานของ OOP ได้แก่ Class, Object, Attribute, Method
- เข้าใจหลักการสำคัญของ OOP ได้แก่ Encapsulation, Inheritance, Polymorphism, Abstraction

ภาษาเชิงโครงสร้างและภาษาเชิงวัตถุ

- ภาษาเชิงโครงสร้าง (Structure-Oriented Programming)
  - เน้นลำดับการทำงานและฟังก์ชัน (Functions/Procedures)
  - ข้อมูลและการประมวลผลแยกจากกัน
  - เหมาะสำหรับโปรแกรมขนาดเล็กและไม่ซับซ้อน
  - การจัดการโปรแกรมขนาดใหญ่ทำได้ยาก, การแก้ไขโค้ดส่งผลกระทบวงกว้าง
- ภาษาเชิงวัตถุ (Object-Oriented Programming)
  - เน้นการรวมข้อมูลและการประมวลผลเข้าด้วยกันเป็น “วัตถุ”
  - สร้างโมเดลที่สะท้อนโลกแห่งความเป็นจริง
  - แก้ปัญหาความซับซ้อนของโปรแกรมขนาดใหญ่

การเขียนโปรแกรมเชิงวัตถุ

ทำไมต้อง OOP ?

- **แก้ปัญหาความซับซ้อน:** จัดการโปรแกรมขนาดใหญ่ได้ง่ายขึ้น
- **นำกลับมาใช้ใหม่ได้ (Reusability):** โค้ดที่เขียนไว้สามารถนำไปใช้ซ้ำได้
- **บำรุงรักษาง่าย (Maintainability):** การเปลี่ยนแปลงส่วนหนึ่งไม่กระทบส่วนอื่นมากนัก
- **ขยายระบบได้ง่าย (Scalability):** เพิ่มฟังก์ชันการทำงานใหม่ๆ ได้สะดวก
- **ความปลอดภัยของข้อมูล (Data Security):** ควบคุมการเข้าถึงข้อมูลได้ดีขึ้น

ส่วนประกอบของ OOP

- Class (คลาส)
- Object (วัตถุ)
- Attribute (คุณสมบัติ)
- Method (เมธอด)

ส่วนประกอบของ OOP : Class

- **Class (คลาส) คืออะไร?**
- แผนพิมพ์เขียว (Blueprint) หรือแม่แบบสำหรับวัตถุ (object)
- กำหนดโครงสร้างและพฤติกรรมของวัตถุ
- <u>ตัวอย่างเชิงเปรียบเทียบ</u> Class คือ 'รถยนต์' กำหนดว่ารถยนต์มีคุณสมบัติอะไรบ้าง (ยี่ห้อ, สี, รุ่น) และทำอะไรได้บ้าง (วิ่ง, เบรก, เลี้ยว)

ส่วนประกอบของ OOP : Class

ตัวอย่างโค้ด Java

```
// Car.java (Class)
public class Car {
    // Attributes (ข้อมูล)
    String brand;
    String model;
    String color;

    // Methods (พฤติกรรม)
    public void start() {
        System.out.println(brand + " " + model + " is starting.");
    }

    public void stop() {
        System.out.println(brand + " " + model + " is stopping.");
    }
}
```

ส่วนประกอบของ OOP : Object

- **Object (วัตถุ) คืออะไร?**
- สิ่งที่ถูกสร้างขึ้นมาจาก Class (Instance of a Class)
- มีคุณสมบัติ (Attributes) และสามารถกระทำบางอย่าง (Methods) ได้จริง
- แต่ละ Object มีสถานะ (State) และพฤติกรรม (Behavior) เป็นของตัวเอง
- <u>ตัวอย่างเชิงเปรียบเทียบ</u> จาก Class 'รถยนต์' เราสามารถสร้าง Object 'รถยนต์ของฉัน' (Honda Civic สีแดง) หรือ 'รถยนต์เพื่อน' (Toyota Camry สีดำ)

ส่วนประกอบของ OOP : Object

ตัวอย่างโค้ด Java

```
// Main.java (Creating Objects)
public class Main {
   public static void main(String[] args) {
      // Creating objects (instances) of the Car class
      Car myCar = new Car(); // Object 1
      myCar.brand = "Honda";
      myCar.model = "Civic";
      myCar.color = "Red";

      Car yourCar = new Car(); // Object 2
      yourCar.brand = "Toyota";
      yourCar.model = "Camry";
      yourCar.color = "Black";

      // Calling methods on objects
      myCar.start();
      yourCar.stop();
   }
}
```

ส่วนประกอบของ OOP : Attribute

- **Attribute (คุณสมบัติ) คืออะไร?**
- ข้อมูลที่บอกสถานะหรือคุณลักษณะของ Object นั้นๆ
- เปรียบเสมือนคุณสมบัติทางกายภาพของวัตถุ (เช่น สี, ขนาด, ยี่ห้อ)
- ในโค้ด Java คือ ตัวแปร (Variables) ที่ประกาศใน Class

ส่วนประกอบของ OOP : Method

- **Method (เมธอด) คืออะไร?**
- การกระทำหรือพฤติกรรมที่ Object นั้นๆ สามารถทำได้
- เปรียบเสมือนสิ่งที่วัตถุนั้นทำได้ (เช่น เดิน, พูด, ขับ)
- ในโค้ด Java คือ ฟังก์ชัน (Functions) ที่ประกาศใน Class

ส่วนประกอบของ OOP : Attribute & Method

ตัวอย่างโค้ด Java

```
public class Car {
   // Attributes (ข้อมูล/คุณสมบัติ)
   String brand;  // ยี่ห้อ
   String model;  // รุ่น
   String color;  // สี

   // Methods (พฤติกรรม/การกระทำ)
   public void start() { // เมธอด "สตาร์ท"
      System.out.println(brand + " " + model + " is starting.");
   }

   public void stop() { // เมธอด "หยุด"
      System.out.println(brand + " " + model + " is stopping.");
   }

   public void accelerate(int speed) { // เมธอด "เร่งความเร็ว" พร้อมรับค่า
      System.out.println(brand + " " + model + " is accelerating to " + speed + " km/h.");
   }
}
```

หลักการสำคัญของ OOP

- <u>Encapsulation</u> การห่อหุ้มข้อมูลและเมธอดเข้าด้วยกัน และควบคุมการเข้าถึง
- <u>Inheritance</u> การสืบทอดคุณสมบัติจาก Class แม่สู่ Class ลูก
- <u>Polymorphism</u> การที่วัตถุสามารถแสดงพฤติกรรมได้หลากหลายรูปแบบ
- <u>Abstraction</u> การซ่อนรายละเอียดที่ซับซ้อน แสดงเฉพาะสิ่งที่จำเป็น

หลักการสำคัญของ OOP : Encapsulation

- <u>Encapsulation(การห่อหุ้ม)</u>
- การรวมข้อมูล (Attributes) และเมธอด (Methods) ที่เกี่ยวข้องเข้าไว้ด้วยกันภายใน Class เดียวกัน
- ซ่อนรายละเอียดการทำงานภายใน (Internal Implementation) และควบคุมการเข้าถึงข้อมูลโดยตรงจากภายนอก (การปกป้องข้อมูล)

หลักการสำคัญของ OOP : Encapsulation

- <u>Encapsulation(การห่อหุ้ม)</u>
- **ประโยชน์**
  - ลดความซับซ้อน
  - เพิ่มความปลอดภัยของข้อมูล (Data Hiding)
  - ทำให้การบำรุงรักษาง่ายขึ้น (สามารถเปลี่ยน Logic ภายในได้โดยไม่กระทบภายนอก)
- **การนำไปใช้ใน Java** ใช้ Access Modifiers เช่น private, public

หลักการสำคัญของ OOP : Encapsulation

ตัวอย่างโค้ด Java

```
public class BankAccount {
    private String accountNumber; // ซ่อนข้อมูล account number
    private double balance;       // ซ่อนข้อมูล balance

    public BankAccount(String accNum, double initialBalance) {
        this.accountNumber = accNum;
        this.balance = initialBalance;
    }

    // Public method to deposit money (controlled access)
    public void deposit(double amount) {
        if (amount > 0) {
            this.balance += amount;
            System.out.println("Deposited: " + amount + ". New balance: " + this.balance);
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }
```

หลักการสำคัญของ OOP : Encapsulation

ตัวอย่างโค้ด Java (ต่อ)

```
    // Public method to withdraw money (controlled access)
    public void withdraw(double amount) {
        if (amount > 0 && this.balance >= amount) {
            this.balance -= amount;
            System.out.println("Withdrew: " + amount + ". New balance: " + this.balance);
        } else {
            System.out.println("Invalid withdrawal amount or insufficient funds.");
        }
    }

    // Public method to get balance (controlled access)
    public double getBalance() { // Getter method
        return this.balance;
    }

    // Public method to get account number (controlled access, but can't be set directly)
    public String getAccountNumber() { // Getter method
        return this.accountNumber;
    }
}
```

หลักการสำคัญของ OOP : Inheritance

- <u>Inheritance(การสืบทอด)</u>
- การที่ Class หนึ่ง (Subclass/Child Class) สามารถสืบทอดคุณสมบัติ (Attributes) และพฤติกรรม (Methods) มาจากอีก Class หนึ่ง (Superclass/Parent Class) ได้
- **ประโยชน์**
  - **การนำโค้ดกลับมาใช้ใหม่ (Code Reusability)** ไม่ต้องเขียนโค้ดซ้ำ
  - สร้างความสัมพันธ์แบบ "is-a" (เป็น... เช่น 'Dog is an Animal')
  - ขยายความสามารถของ Class เดิม
- **การนำไปใช้ใน Java** ใช้คีย์เวิร์ด extends

หลักการสำคัญของ OOP : Inheritance

ตัวอย่างโค้ด Java

```
// Superclass / Parent Class
public class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }

    public void sleep() {
        System.out.println(name + " is sleeping.");
    }
}
```

หลักการสำคัญของ OOP : Inheritance

ตัวอย่างโค้ด Java

```
// Subclass / Child Class
public class Dog extends Animal {
   public Dog(String name) {
      super(name); // Call parent class constructor
   }

   public void bark() {
      System.out.println(name + " is barking.");
   }
}

// Main to demonstrate inheritance
public class Zoo {
   public static void main(String[] args) {
      Dog myDog = new Dog("Buddy");
      myDog.eat();  // Inherited method from Animal
      myDog.sleep(); // Inherited method from Animal
      myDog.bark(); // Dog's specific method
   }
}
```

หลักการสำคัญของ OOP : Polymorphism

- <u>Polymorphism(การพ้องรูป / หลากหลายรูปแบบ)</u>
- ความสามารถของวัตถุที่จะถูกปฏิบัติต่อในหลายรูปแบบ (One Interface, Multiple Implementations)
- วัตถุประเภทเดียวกันสามารถตอบสนองต่อคำสั่งเดียวกันในรูปแบบที่แตกต่างกันได้
- **ประโยชน์**
  - เพิ่มความยืดหยุ่นของโค้ด
  - ลดการใช้ if-else หรือ switch-case ซับซ้อน
  - ทำให้โค้ดอ่านง่ายและบำรุงรักษาง่ายขึ้น

หลักการสำคัญของ OOP : Polymorphism

- การนำไปใช้ใน Java
  - **Method Overriding** Subclass เขียนทับเมธอดของ Superclass (ตอน Runtime)
  - **Method Overloading** เมธอดชื่อเดียวกันแต่รับพารามิเตอร์ต่างกัน (ตอน Compile-time)
  - **Interface** (จะกล่าวถึงในบทต่อๆ ไป)

หลักการสำคัญของ OOP : Polymorphism

ตัวอย่างโค้ด Java

```
// Superclass
public class Shape {
    public void draw() {
        System.out.println("Drawing a generic shape.");
    }
}

// Subclass 1
public class Circle extends Shape {
    @Override // Annotation indicating method overriding
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
}
```

หลักการสำคัญของ OOP : Polymorphism

ตัวอย่างโค้ด Java

```
// Subclass 2
public class Rectangle extends Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Rectangle.");
   }
}

// Main to demonstrate polymorphism
public class DrawingApp {
   public static void main(String[] args) {
      Shape s1 = new Circle();    // s1 is a Shape, but actually a Circle
      Shape s2 = new Rectangle(); // s2 is a Shape, but actually a Rectangle
      Shape s3 = new Shape();     // s3 is a Shape

      s1.draw(); // Calls Circle's draw()
      s2.draw(); // Calls Rectangle's draw()
      s3.draw(); // Calls Shape's draw()
   }
}
```

END.

Q & A