

Object-Oriented Programming (OOP)



Object-Oriented Programming (OOP) หรือ การเขียนโปรแกรมเชิงวัตถุ คือ รูปแบบหรือแนวคิดในการเขียนโปรแกรมคอมพิวเตอร์ โดยมองโค้ดให้เป็นเสมือน "วัตถุ" (Object) ขึ้นหนึ่งในโลกความเป็นจริง แทนที่จะเขียนเป็นลำดับขั้นตอนหรือฟังก์ชันแบบเดิม

องค์ประกอบสำคัญของ OOP

- **Class (คลาส):** เปรียบเสมือน "พิมพ์เขียว" หรือแบบแปลนที่กำหนดคุณลักษณะและพฤติกรรม เช่น คลาส Car (รถยนต์)
- **Object (วัตถุ):** คือสิ่งที่ถูกสร้างขึ้น (Instance) มาจาก Class นั้น ๆ เช่น รถยนต์คันสีแดงทะเบียน กข 123 ซึ่งเป็นรูปธรรมขึ้นมา
- **Attributes (คุณลักษณะ):** ข้อมูลหรือตัวแปรประจำวัตถุ เช่น สี ยี่ห้อ ความเร็ว
- **Methods (พฤติกรรม/การกระทำ):** ฟังก์ชันหรือการทำงานที่วัตถุนั้นทำได้ เช่น startEngine(), accelerate(), brake()



องค์ประกอบสำคัญของ OOP

4 เสาหลักของ OOP (4 Pillars)

การเขียนโปรแกรมเชิงวัตถุอาศัยหลักการสำคัญ 4 ประการนี้เพื่อให้โค้ดเป็นระเบียบและนำกลับมาใช้ใหม่ได้ง่าย:

- Encapsulation (การห่อหุ้มข้อมูล):** การซ่อนรายละเอียดภายในที่ไม่จำเป็น และควบคุมการเข้าถึงข้อมูล (เช่น ใช้ตัวแปรแบบ private ร่วมกับฟังก์ชัน getter/setter) เพื่อความปลอดภัยของข้อมูล
- Abstraction (การซ่อนความซับซ้อน/นามธรรม):** การแสดงเฉพาะส่วนที่จำเป็นต้องใช้ให้ผู้พัฒนาเห็น โดยซ่อนโค้ดเบื้องหลังที่ซับซ้อนไว้ (เช่น กดปุ่มขับรถ drive() โดยไม่ต้องสนใจว่าเครื่องยนต์จะเปิดอย่างไร)
- Inheritance (การสืบทอดคุณสมบัติ):** การสร้างคลาสใหม่ (Child Class) โดยสืบทอดคุณสมบัติและพฤติกรรมมาจากคลาสแม่ (Parent Class) เพื่อลดการเขียนโค้ดซ้ำซ้อน เช่น คลาส ElectricCar สืบทอดมาจากคลาส Car

4. Polymorphism (การพ้องรูป / แปลงร่าง): ความสามารถของเมธอดชื่อเดียวกัน แต่สามารถทำงานหรือแสดงพฤติกรรมที่ต่างกันได้ตามแต่ละคลาสลูก (เช่น เมธอด speak() ของคลาสสัตว์ แต่สุนัขส่งเสียง "โ으่ง" แมวส่งเสียง "เหมียว")

oop 4 เสาหลักของ OOP

(4 Pillars of Object-Oriented Programming)

ทำให้โค้ดเป็นระเบียบ
นำกลับมาใช้ใหม่ได้ง่าย

4 เสาหลักนี้
คือหัวใจของ OOP
เลยนะคะ

1 Encapsulation (การห่อหุ้มข้อมูล)

การซ่อนรายละเอียดภายในที่ไม่จำเป็น และควบคุมการเข้าถึงข้อมูล (เช่น ใช้ตัวแปรแบบ private ร่วมกับ ฟังก์ชัน getter/setter) เพื่อความปลอดภัยของข้อมูล

```
class User {  
    private String name;  
    private String email;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

ข้อมูลถูกปกป้อง
ควบคุมการเข้าถึงได้

2 Abstraction (การซ่อนความซับซ้อน/นามธรรม)

การแสดงเฉพาะส่วนที่จำเป็นต้องใช้ให้ผู้พัฒนาเห็น โดยซ่อนโค้ดเบื้องหลังที่ซับซ้อนไว้ (เช่น กดปุ่มขับรถ drive()) โดยไม่ต้องสนใจว่าเครื่องยนต์จุดระเบิดอย่างไร

แค่เรียกใช้
เมธอด drive()

ซ่อนรายละเอียดที่ซับซ้อนไว้ภายใน

drive()

ผู้ใช้โฟกัสที่สิ่งที่จำเป็น
ไม่ต้องกังวลเรื่องความซับซ้อน

3 Inheritance (การสืบทอดคุณสมบัติ)

การสร้างคลาสใหม่ (Child Class) โดยสืบทอดคุณสมบัติและพฤติกรรมมาจากคลาสแม่ (Parent Class) เพื่อลดการเขียนโค้ดซ้ำซ้อน เช่น คลาส ElectricCar สืบทอดมาจากคลาส Car

Parent Class Car

- brand
- color
- speed
- + startEngine()

สืบทอด
คุณสมบัติ + พฤติกรรม

Child Class ElectricCar

- brand
- color
- speed
- + batteryLevel
- + startEngine()

ช่วยลดการเขียนโค้ดซ้ำซ้อน
และทำให้ดูแลโค้ดได้ง่ายขึ้น

4 Polymorphism (การพ้องรูป / แปลงร่าง)

ความสามารถของเมธอดชื่อเดียวกัน แต่สามารถทำงานหรือแสดงพฤติกรรมที่ต่างกันได้ตามแต่ละคลาสลูก (เช่น เมธอด speak() ของคลาสสัตว์ แต่สุนัขส่งเสียง "โ으่ง" แมวส่งเสียง "เหมียว")

เมธอดเดียวกัน : speak()

โ으่ง!

เหมียว!

จิบๆ!

Dog speak()

Cat speak()

Bird speak()

ทำให้โค้ดยืดหยุ่น รองรับการทำงาน
และปรับเปลี่ยนได้ง่าย

OOP = โค้ดที่ดีขึ้น • อ่านง่าย • ดูแลรักษาง่าย • ขยายได้ในอนาคต

เขียนโปรแกรม
ให้เหมือนสร้างโลกใบเล็ก ๆ
ในคอมพิวเตอร์

Functional Programming

Functional Programming คือ อะไร

ที่اماที่ไป เกี่ยวกับ เรื่องของ โครงสร้าง Functional Programming , โดยหลักการทำงานของ Function Programming นั้น มีประโยชน์ในการทำงานอย่างไรบ้าง ไม่ว่าจะเป็นภาษา [PHP](#), [Python](#), [C#](#) ก็สามารถเขียนในรูปแบบของ Function ได้เช่นกัน, จริงๆแล้ว Functional Programming นั้นเกิดมาเป็นเวลานาน โดยพัฒนามาจาก Lambda calculus

Functional Programming (FP) หรือการเขียนโปรแกรมเชิงฟังก์ชัน คือรูปแบบหรือกระบวนทัศน์ (Paradigm) ในการเขียนโปรแกรมที่เน้นการสร้างซอฟต์แวร์ผ่านการประยุกต์ใช้และการประกอบกันของ "ฟังก์ชันทางคณิตศาสตร์" คล้ายกับการคำนวณในวิชาคณิตศาสตร์ โดยหลีกเลี่ยงการเปลี่ยนแปลงข้อมูลและสถานะของโปรแกรม

λ Functional Programming (FP)
การเขียนโปรแกรมเชิงฟังก์ชัน

Functional Programming (FP) คือ รูปแบบหรือกระบวนทัศน์ (Paradigm) ในการเขียนโปรแกรมที่เน้นการสร้างซอฟต์แวร์ผ่านการประยุกต์ใช้และการประกอบกันของ "ฟังก์ชันทางคณิตศาสตร์" โดยหลีกเลี่ยงการเปลี่ยนแปลงข้อมูลและสถานะของโปรแกรม

แนวคิดหลักของ FP

- ใช้ฟังก์ชันเป็นหลักเขียนโปรแกรมโดยการประกอบกันของฟังก์ชัน
- ไม่เปลี่ยนแปลงข้อมูล (Data) ไม่ถูกแก้ไข แต่จะสร้างข้อมูลใหม่แทน (Immutable)
- ไม่มีสถานะ (State) หลีกเลี่ยงการใช้ตัวแปรที่มีสถานะ (เช่น ตัวแปรแบบ Global)
- เน้นความบริสุทธิ์ของฟังก์ชัน ฟังก์ชันควรให้ผลลัพธ์เดิมเสมอเมื่อรับข้อมูลเดิม

ตัวอย่างการทำงาน

```
// ฟังก์ชันแบบ Functional Programming
const numbers = [1, 2, 3, 4, 5];
const double = x => x * 2;
const addOne = x => x + 1;

const result = numbers
  .map(double) // 2, 4, 6, 8, 10
  .map(addOne); // 3, 5, 7, 9, 11
console.log(result); // [3, 5, 7, 9, 11]
```

จุดเด่นของ FP

- ✓ โค้ดอ่านง่าย และกระชับ
- ✓ ลดข้อผิดพลาดจากการเปลี่ยนแปลงข้อมูล
- ✓ ง่ายต่อการทดสอบ (Testing)
- ✓ สามารถทำงานแบบขนาน (Parallel) ได้ดี
- ✓ เหมาะกับการประมวลผลข้อมูลขนาดใหญ่ และระบบที่ต้องการความเสถียร

เหมาะกับการเน้นการประมวลผลข้อมูล และการคำนวณ เช่น Data Processing, AI, Machine Learning, Big Data

เปรียบเทียบแบบง่าย ๆ

Imperative (แบบเดิม)
 $x = 5$
 $x = x + 1$
เน้นการเปลี่ยนแปลงข้อมูล และสถานะของโปรแกรม

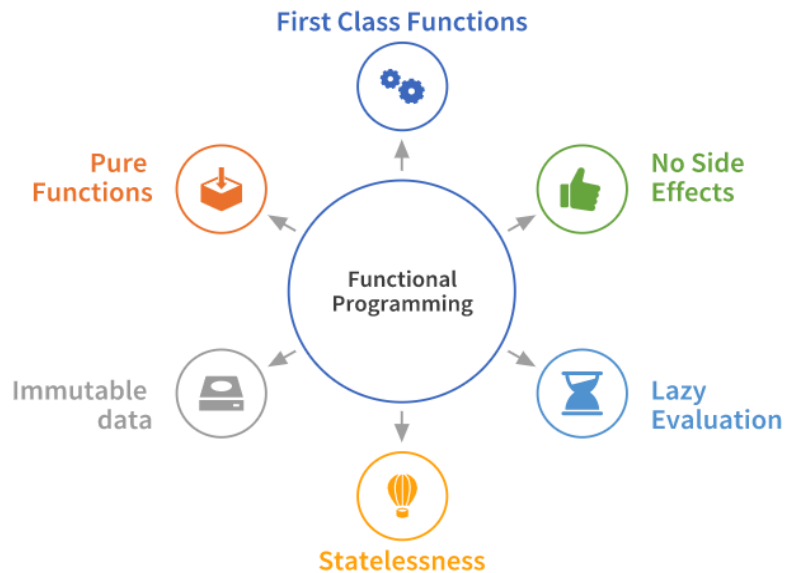
Functional (FP)
addOne(5) // 6
เน้นการใช้ฟังก์ชัน และไม่เปลี่ยนแปลงข้อมูลเดิม

ตัวอย่างการใช้งาน

- การประมวลผลข้อมูลขนาดใหญ่ (Big Data)
- ปัญญาประดิษฐ์ (AI / Machine Learning)
- เว็บแอปพลิเคชัน (React, Vue, etc.)
- แอปพลิเคชันบนมือถือ (React Native)

Functional Programming
Clean Code
Better Software

λ ... คิดแบบฟังก์ชัน • เขียนแบบสะอาด • สร้างซอฟต์แวร์ที่ดีกว่า



Mind map Functional Programming

ในการเขียน Function, เราจะเน้นไปเขียนรูปแบบ Pure Function มากกว่า, ซึ่งหลักการทำงานจริงๆ ก็มีอยู่ 3 อย่างด้วยกัน

1. รับค่าเข้ามา
2. ดำเนินการบางอย่างเกี่ยวกับค่านั้น
3. Return ผลลัพธ์ออกไป

แนวคิดหลัก (Core Concepts)

- Pure Function (ฟังก์ชันบริสุทธิ์): ฟังก์ชันที่เมื่อรับค่าอินพุตเดิมเข้ามา จะให้ค่าเอาต์พุตเดิมเสมอ และไม่ มีผลข้างเคียง (Side Effects) เช่น ไม่ไปเปลี่ยนแปลงค่าตัวแปรภายนอกหรือฐานข้อมูล [1]
- Immutability (ความไม่เปลี่ยนแปลง): ข้อมูลหรือตัวแปรเมื่อถูกสร้างขึ้นมาแล้ว จะไม่สามารถแก้ไขค่า เดิมได้ หากต้องการเปลี่ยนข้อมูล จะต้องสร้างข้อมูลชุดใหม่ขึ้นมาแทน [1, 2]
- First-Class Functions (ฟังก์ชันระดับเฟิร์สคลาส): มองฟังก์ชันเป็นค่า (Value) ชนิดหนึ่ง สามารถนำไป กำหนดให้ตัวแปร ส่งเป็นอาร์กิวเมนต์ หรือให้ฟังก์ชันอื่นรีเทิร์นออกมาได้

- Higher-Order Functions (ฟังก์ชันลำดับสูง): ฟังก์ชันที่สามารถรับฟังก์ชันอื่นเข้ามาเป็นพารามิเตอร์ หรือส่งฟังก์ชันออกไปเป็นผลลัพธ์ได้ เช่น map, filter, และ reduce

ประโยชน์ของ Functional Programming

- ลดข้อผิดพลาด (Fewer Bugs): เนื่องจากไม่มีการแชร์ตัวแปรที่เปลี่ยนแปลงได้ (Mutable State) จึงช่วยลดปัญหาค่าข้อมูลเพี้ยนจาก Side Effects
- ทดสอบง่าย (Easy to Test): เพราะ Pure Function ให้ผลลัพธ์ที่แน่นอนตามอินพุต จึงเขียน Unit Test ได้ง่ายมาก
- รองรับการทำงานพร้อมกัน (Concurrency/Parallelism): ข้อมูลไม่ถูกแก้ไขระหว่างทาง ทำให้ปลอดภัยในการรันหลายเธรด (Threads) พร้อมกัน
- อ่านและประกอบโค้ดได้ง่าย: สามารถนำฟังก์ชันเล็กๆ มาประกอบเข้าด้วยกัน (Function Composition) เพื่อสร้างฟังก์ชันที่ซับซ้อนขึ้นได้

เหตุผลที่เราต้องพูดถึงเรื่อง Functional Programming กัน ?

เนื่องจากในปัจจุบันมีการเปลี่ยนแปลงมากมายเกิดขึ้น , ยกตัวอย่างจำนวนข้อมูลมีขนาดใหญ่มากขึ้น (Big data), ดังเราต้องการแนวทางที่สามารถเขียน Programming, เพื่อจัดการและประมวลผล ที่มีขนาดใหญ่ ให้เกิดความสมดุล มากขึ้น และรองรับการขยายตัวได้ง่ายและเหมาะสมมากขึ้น

เปรียบเทียบกับ Object-Oriented Programming (OOP)

- OOP (เชิงวัตถุ): เน้นการจัดกลุ่มข้อมูลและพฤติกรรมไว้ด้วยกันในรูปของ "Object" ซึ่งมีการเก็บสถานะ (State) ที่เปลี่ยนแปลงได้ภายในตัววัตถุนั้นๆ
- FP (เชิงฟังก์ชัน): เน้นการแปลงข้อมูล (Data Transformation) ผ่านลำดับของฟังก์ชัน โดยแยกข้อมูลออกจากตัวฟังก์ชัน และหลีกเลี่ยงการเปลี่ยนสถานะ

ตัวอย่างการเขียนโค้ดเปรียบเทียบระหว่าง Object-Oriented Programming (OOP) และ Functional Programming (FP) โดยใช้ภาษา JavaScript คำนวณราคาสินค้าในตะกร้าช้อปปิ้ง

โจทย์: เรามีรายการสินค้าที่มีราคาระบุไว้ เราต้องการกรองเอาเฉพาะสินค้าที่มีราคาตั้งแต่ 20 บาทขึ้นไป จากนั้นบวกภาษี (VAT) 7% เข้าไปในแต่ละชิ้น และหาผลรวมราคาสินค้าทั้งหมดในตะกร้า

1. วิธีเขียนแบบ Object-Oriented Programming (OOP) OOP จะเน้นการสร้าง Object (วัตถุ) เพื่อเก็บข้อมูล (Data) และ พฤติกรรม (Method) ไว้ด้วยกันในรูปของคลาส (Class) และมีการเปลี่ยนแปลงสถานะภายในวัตถุ (Mutable State)

javascript

```
class ShoppingCart {
  constructor() {
    this.items = []; // เก็บสถานะของข้อมูลไว้ใน Object
  }

  addItem(price) {
    this.items.push(price);
  }

  // เมธอดที่ทำหน้าที่คำนวณและเปลี่ยนสถานะภายในวัตถุ
  calculateTotal() {
    let total = 0;
    for (let i = 0; i < this.items.length; i++) {
      // 1. กรองราคาสินค้าตั้งแต่ 20 บาทขึ้นไป
      if (this.items[i] >= 20) {
        // 2. บวกภาษี 7% และ 3. หาผลรวม
        total += this.items[i] * 1.07;
      }
    }
    return total;
  }
}

// การใช้งาน
const myCart = new ShoppingCart();
myCart.addItem(10);
myCart.addItem(25);
myCart.addItem(50);

console.log(myCart.calculateTotal()); // ผลลัพธ์: 80.25
```



2. วิธีเขียนแบบ Functional Programming (FP)

FP จะแยกข้อมูลออกจากฟังก์ชันอย่างเด็ดขาด ข้อมูลจะไม่ถูกแก้ไข (Immutability) แต่จะใช้วิธี ส่งข้อมูลผ่านท่อของฟังก์ชัน (**Data Pipeline**) โดยการประกอบกันของฟังก์ชันสำเร็จรูป (Function Composition)

javascript

```
// 1. ข้อมูลดิบที่เป็น Array ธรรมดา (ไม่ผูกกับเมธอดใดๆ)
const cartItems =;

// 2. สร้าง Pure Functions ย่อยๆ ที่ทำงานเฉพาะทาง
const isExpensiveThan20 = (price) => price >= 20;
const addTax = (price) => price * 1.07;
const sum = (acc, price) => acc + price;

// 3. ใช้ Higher-Order Functions (filter, map, reduce) มาร้อยเรียงกัน
const calculateTotal = (items) =>
  items
    .filter(isExpensiveThan20) // กรองตัวเลข >= 20
    .map(addTax)                // คำนวณภาษีเพิ่ม 7% ในทุกตัวที่เหลือ
    .reduce(sum, 0);           // รวมผลลัพธ์ทั้งหมดเข้าด้วยกัน

// การใช้งาน
const total = calculateTotal(cartItems);

console.log(total); // ผลลัพธ์: 80.25
console.log(cartItems); // [10, 25, 50] ข้อมูลดิบยังอยู่ครบ ไม่ถูกเปลี่ยนแปลง (Immutability)
```

Use code with caution.

