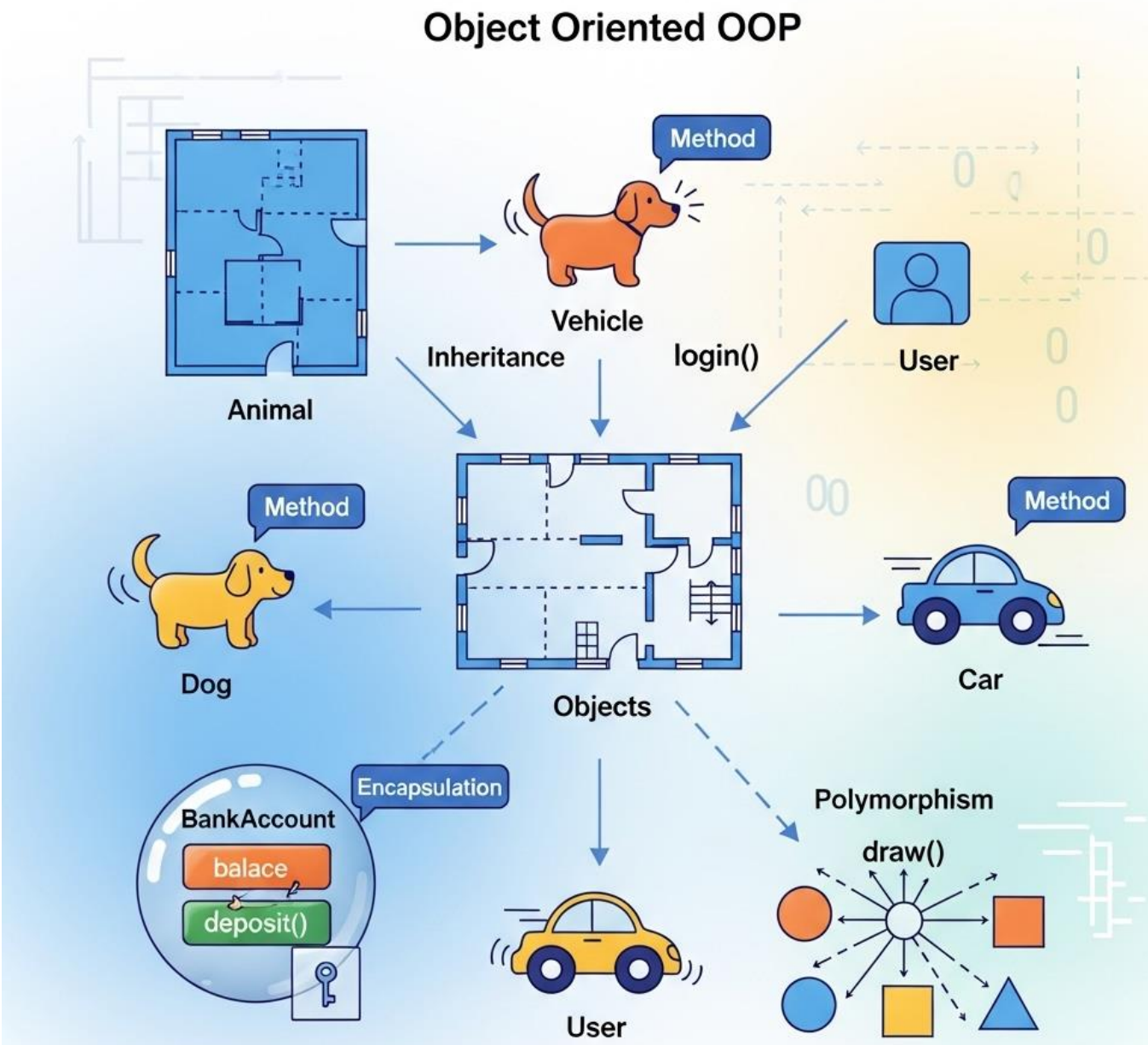
Object Oriented Programming in Python

Shutchon Premchaisawatt

OOP คือ

OOP คือ **รูปแบบการเขียนโปรแกรม** รูปแบบหนึ่ง ที่จะมองทุกอย่างเป็นเหมือน "วัตถุ" (Object) ในโลกจริงค่ะ โดยแต่ละวัตถุจะมีการเก็บ **ข้อมูล (attributes)** และ **การกระทำ (methods)** ของตัวเองไว้ด้วยกัน ทำให้โค้ดของเราเป็นระเบียบอ่านง่าย และนำกลับมาใช้ใหม่ได้สะดวกขึ้นมากเลยค่ะ



ตัวอย่าง "พิมพ์เขียวสร้างบ้าน" นะคะ

- **คลาส (Class):** ก็คือ "พิมพ์เขียว" นั่นเองค่ะ มันจะกำหนดโครงสร้างว่าบ้านต้องมีอะไรบ้าง เช่น ต้องมีกี่ห้องนอน (ข้อมูล) และทำอะไรได้บ้าง เช่น เปิดไฟได้ (การกระทำ) แต่ตัวพิมพ์เขียวเองยังไม่ใช่บ้านจริงๆ นะคะ
- **อ็อบเจกต์ (Object):** คือ "บ้าน" ที่สร้างเสร็จแล้วจากพิมพ์เขียวนั้นๆ ค่ะ เราสามารถใช้พิมพ์เขียวอันเดียวสร้างบ้านกี่หลังก็ได้ ซึ่งบ้านแต่ละหลัง (แต่ละ Object) ก็จะมีสถานะเป็นของตัวเอง เช่น บ้านหลังหนึ่งอาจจะเปิดไฟอยู่ ในขณะที่อีกหลังปิดไฟอยู่ก็ได้ค่ะ

สร้างพิมพ์เขียว

งั้นเรามาสร้าง Class และ Object แรกของเรากันเลยนะคะ

เราจะลองสร้าง "พิมพ์เขียว" ของน้องหมากันค่ะ โดยจะกำหนดว่าน้องหมาทุกตัวต้องมี **ชื่อ (name)** และ **สายพันธุ์ (breed)** และมีความสามารถในการ **เห่า (bark)** ค่ะ

```
class Dog:
    # นี่คือ "พิมพ์เขียว" ตอนสร้างน้องหมาตัวใหม่ขึ้นมา
    # จะถูกเรียกใช้ทันทีที่เราสร้าง Object ใหม่
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
        print(f"น้องหมาตัวใหม่ถือกำเนิดแล้ว! ชื่อ {self.name}")

    # นี่คือ "การกระทำ" ที่น้องหมาทำได้
    def bark(self):
        print(f"{self.name} บอกว่า: โฮ่ง! โฮ่ง!")
```

อธิบาย

•class Dog: คือการบอก Python ว่าเรากำลังจะสร้างพิมพ์เขียวใหม่ชื่อว่า Dog ค่ะ

•def __init__(self, name, breed): เป็นเมธอดพิเศษที่เรียกว่า **constructor** (คอนสตรัคเตอร์) ค่ะ มันจะทำงาน **อัตโนมัติ** ทุกครั้งที่เราสร้างน้องหมาตัวใหม่ (สร้าง Object) ขึ้นมา

•self จะหมายถึง "ตัวของ Object เอง" ที่กำลังจะถูกสร้างขึ้นมาค่ะ

•self.name = name คือการนำค่า name ที่เราป้อนเข้ามา มาเก็บเป็น "ข้อมูล" ของน้องหมาตัวนั้นๆ

•def bark(self): คือเมธอด (การกระทำ) ที่เราสร้างขึ้นมาเองค่ะ สังเกตว่าต้องมี self อยู่ในวงเล็บเสมอ เพื่อให้มันรู้ว่าจะต้องทำงานกับ Object ตัวไหน

```
class Dog:
    # นี่คือ "พิมพ์เขียว" ตอนสร้างน้องหมาตัวใหม่ขึ้นมา
    # จะถูกเรียกใช้ทันทีที่เราสร้าง Object ใหม่
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
        print(f"น้องหมาตัวใหม่ถือกำเนิดแล้ว! ชื่อ {self.name}")

    # นี่คือ "การกระทำ" ที่น้องหมาทำได้
    def bark(self):
        print(f"{self.name} บอกว่า: โฮ่ง! โฮ่ง!")
```

สร้างและทดสอบ Object

```
# สร้างน้องหมาตัวแรกชื่อ "บัดดี้" สายพันธุ์ "โกลเด้น"
dog1 = Dog("บัดดี้", "โกลเด้น")

# สร้างน้องหมาอีกตัวชื่อ "ลัคกี้" สายพันธุ์ "ชิบะ"
dog2 = Dog("ลัคกี้", "ชิบะ")

# ลองเรียกใช้ "ข้อมูล" และ "การกระทำ" ของแต่ละตัว
print(f"น้องหมาตัวแรกของฉันชื่อ {dog1.name}") # เข้าถึงข้อมูล .name
dog2.bark() # เรียกใช้การกระทำ .bark()
```

ผลลัพธ์บน console:

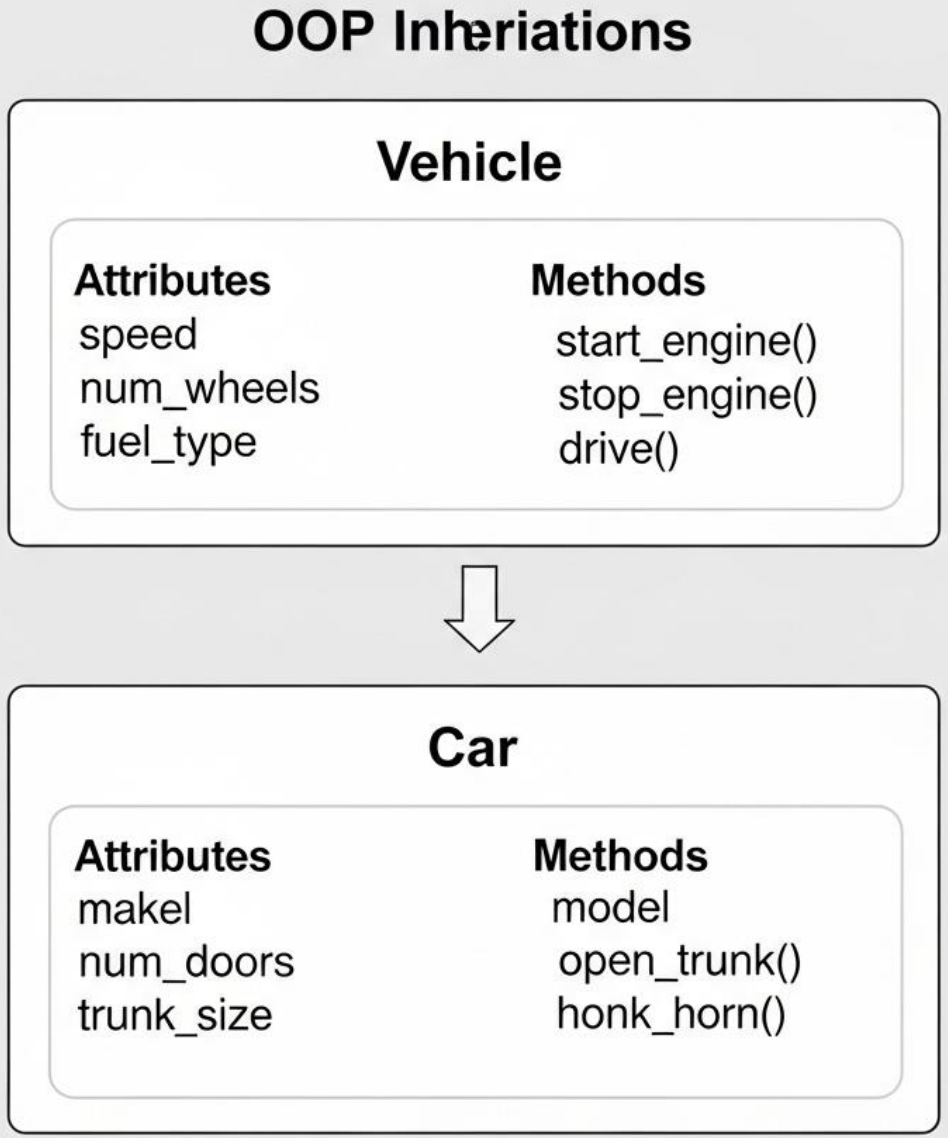
น้องหมาตัวใหม่ถือกำเนิดแล้ว! ชื่อ บัดดี้ น้องหมาตัวใหม่ถือกำเนิดแล้ว! ชื่อ ลัคกี้ น้องหมาตัวแรกของฉันชื่อ บัดดี้ ลัคกี้ บอกว่า: โฮ่ง! โฮ่ง!

คำถาม:

print(dog1.breed)คิดว่าผลลัพธ์ที่หน้าจอจะแสดงออกมาเป็นอะไรคะ?

"พลังพิเศษ" ของ OOP ที่เรียกว่า **การสืบทอด (Inheritance)**

- มันจะช่วยให้เราสร้าง Class ใหม่ๆ ได้โดยไม่ต้องเขียนโค้ดซ้ำซ้อนเลยค่ะ
- Inheritance คือหนึ่งในแนวคิดที่ทรงพลังที่สุดของ OOP เลยค่ะ พูดง่ายๆ ก็คือ **การสร้าง Class ใหม่ โดยรับเอาความสามารถ (ข้อมูลและการกระทำ) ทั้งหมดมาจาก Class ที่มีอยู่แล้ว**
- **Parent Class (คลาสแม่):** คือคลาสต้นแบบที่มีความสามารถพื้นฐาน (บางทีก็เรียกว่า Superclass หรือ Base Class)
- **Child Class (คลาสลูก):** คือคลาสใหม่ที่ "สืบทอด" ทุกอย่างมาจากคลาสแม่ แต่สามารถเพิ่มความสามารถพิเศษหรือเปลี่ยนแปลงบางอย่างให้เป็นของตัวเองได้ (บางทีก็เรียกว่า Subclass)



สืบทอด class dog

เรามีคลาส Dog ที่เป็นคลาสแม่ อยู่แล้ว ตอนนี้สมมติว่าเราอยากสร้างคลาสใหม่สำหรับ "สุนัขนำทาง" (GuideDog) ซึ่งสุนัขนำทางก็คือสุนัขประเภทหนึ่ง แต่มีความสามารถพิเศษคือ "การนำทาง" เพิ่มขึ้นมา

class GuideDog(Dog): สังเกตในวงเล็บนะคะ การใส่ (Dog) เข้าไป คือการบอกว่า GuideDog จะเป็นคลาสลูก และจะได้รับความสามารถทั้งหมดมาจาก Dog ค่ะdef lead(self): คือ "การกระทำ" ใหม่ที่เราเพิ่มเข้าไป ซึ่งจะมีเฉพาะในคลาส GuideDog เท่านั้น คลาส Dog ทั่วไปจะไม่มีความสามารถนี้ค่ะ

คำถาม:ถ้าเราสร้าง Object จากคลาสแม่แบบนี้:
normal_dog = Dog("ธรรมดา", "พันธุ์ทาง")
แล้วเราพยายามสั่ง normal_dog.lead()
คิดว่าจะเกิดอะไรขึ้นคะ?

```python
# คลาสแม่ (Parent Class) ที่เราสร้างไว้
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} บอกว่า: โฮ่ง! โฮ่ง!")

# คลาสลูก (Child Class) ที่สืบทอดมาจาก Dog
class GuideDog(Dog):
    # เพิ่มความสามารถใหม่เข้าไป
    def lead(self):
        print(f"{self.name} กำลังนำทางอยู่ครับ!")
```

Encapsulation (การห่อหุ้มข้อมูล)

Encapsulation คือหลักการที่ว่าด้วยการ **"รวมข้อมูล (attributes) และการกระทำ (methods) ที่เกี่ยวข้องไว้ด้วยกันในที่เดียว"** และที่สำคัญคือ **"การซ่อนรายละเอียดภายในและปกป้องข้อมูล"** จากการเข้าถึงโดยตรงจากภายนอกค่ะ

- ลองนึกถึงการขับรถดูนะคะ 🚙
- **Public Interface (สิ่งที่เราใช้งาน):** เรามีพวงมาลัย, คันเร่ง, เบรก, และเกียร์ เราใช้อุปกรณ์เหล่านี้เพื่อ "สั่ง" ให้รถทำงาน โดยไม่จำเป็นต้องรู้เลยว่าเครื่องยนต์ข้างในทำงานอย่างไร หรือมีกลไกซับซ้อนแค่ไหน
- **Private Implementation (สิ่งที่ถูกซ่อนไว้):** คือเครื่องยนต์, ระบบไฟฟ้า, และกลไกต่างๆ ที่ซ่อนอยู่ใต้ฝากระโปรง ผู้ผลิตรถยนต์ออกแบบมาไม่ให้เราไปยุ่งกับส่วนนี้โดยตรง เพื่อป้องกันไม่ให้เราทำอะไรผิดพลาดจนเครื่องพัง
- ในโลกของ OOP ก็เหมือนกันค่ะ **Encapsulation** คือการสร้าง "เกราะ" หรือ "แคปซูล" มาห่อหุ้มข้อมูลสำคัญไว้ และเปิดช่องทาง (Public Methods) ให้คนอื่นมาใช้งานได้อย่างถูกต้องและปลอดภัยเท่านั้น

การใช้งานใน Python

การใช้งานใน Python

ในภาษา Python เราใช้เครื่องหมาย Underscore (_) นำหน้าชื่อ attribute เพื่อบ่งบอกว่ามันเป็น "ของใช้ส่วนตัวภายใน" ค่ะ

- **_variable (Single Underscore):** เป็นแค่ **"ข้อตกลง"** ระหว่างโปรแกรมเมอร์ ว่า "นี่เป็นตัวแปรภายในนะ ไม่ควรยุ่งจากข้างนอก" แต่ก็ยังสามารถเรียกใช้โดยตรงได้อยู่
- **__variable (Double Underscore):** อันนี้จะ **"จริงจัง"** ขึ้นมาหน่อย Python จะทำการ "แปลงชื่อ" (Name Mangling) ของตัวแปรนี้ ทำให้การเรียกใช้โดยตรงจากภายนอกทำได้ยากขึ้นมาก ใช้สำหรับการปกป้องข้อมูลที่ไม่อยากให้ใครมายุ่งจริงๆ

ตัวอย่าง: บัญชีธนาคาร

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner       # public - ใครก็เห็นชื่อเจ้าของบัญชีได้
        self.__balance = balance  # private - ซ่อนยอดเงินไว้ ไม่ให้แก้โดยตรง

    # Public Method สำหรับการ "ฝากเงิน"
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"ฝากเงินสำเร็จ! ยอดเงินใหม่: {self.__balance} บาท")
        else:
            print("ข้อผิดพลาด: จำนวนเงินที่ฝากต้องมากกว่า 0")

    # Public Method สำหรับการ "ถอนเงิน"
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"ถอนเงินสำเร็จ! ยอดเงินใหม่: {self.__balance} บาท")
        else:
            print("ข้อผิดพลาด: ยอดเงินไม่เพียงพอ")

    # Public Method สำหรับการ "ดูยอดเงิน" เท่านั้น
    def get_balance(self):
        print(f"ยอดเงินคงเหลือของคุณ {self.owner} คือ {self.__balance} บาท")
```

มาดู Class BankAccount กันค่ะ เราคงไม่อยากให้ใครมาแก้ตัวเลข balance (ยอดเงิน) ของเราได้ตรงๆ ใช่ไหมคะ

```python
# --- ลองใช้งาน ---
my_account = BankAccount("คิตะ", 1000)

# ใช้งานผ่าน Public Methods ที่ปลอดภัย
my_account.deposit(500)
my_account.withdraw(200)
my_account.get_balance()

# --- ลองแก้ไขยอดเงินโดยตรง (ซึ่งทำไม่ได้!) ---
# my_account.__balance = 999999
# <-- บรรทัดนี้จะทำให้เกิด Error!
# print(my_account.__balance)
# <-- บรรทัดนี้ก็จะ Error เช่นกัน!
```

```
ฝากเงินสำเร็จ! ยอดเงินใหม่: 1500 บาท
ถอนเงินสำเร็จ! ยอดเงินใหม่: 1300 บาท
ยอดเงินคงเหลือของคุณ คิตะ คือ 1300 บาท
```

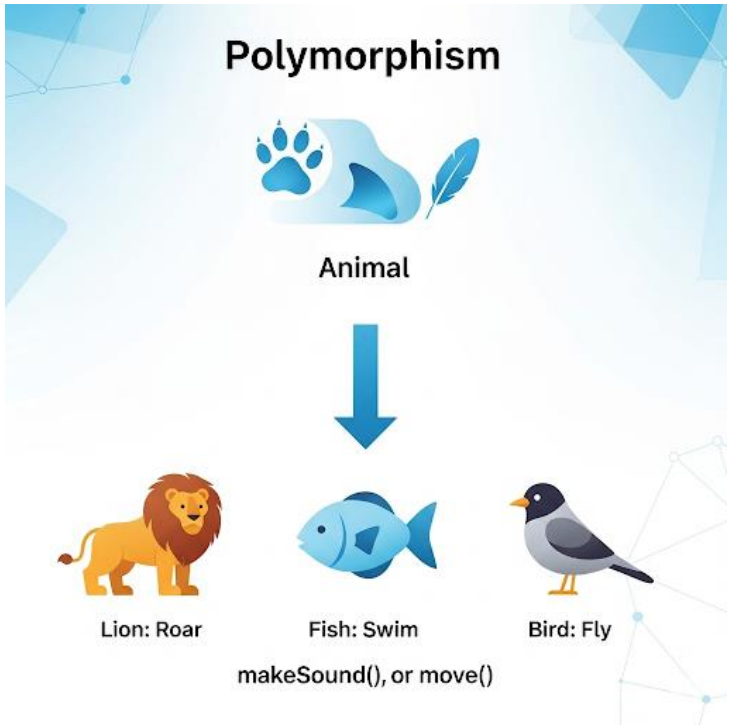
จะเห็นว่าเราไม่สามารถเข้าไปยุ่งกับ __balance โดยตรงได้เลย การจะเปลี่ยนแปลงค่าเงินต้องทำผ่านเมธอด deposit() และ withdraw() ที่เราเตรียมไว้เท่านั้น ซึ่งในเมธอดเหล่านี้เราสามารถใส่เงื่อนไขเพื่อป้องกันข้อมูลที่ผิดพลาดได้ด้วย นี่แหละค่ะคือประโยชน์ของ Encapsulation!

Polymorphism (การพ้องรูป)

คำว่า Polymorphism มาจากภาษากรีก "Poly" (แปลว่า เยอะ) กับ "Morph" (แปลว่า รูปแบบ) รวมกันจึงหมายถึง **"การมีได้หลายรูปแบบ"** ค่ะ

ในโลกของ OOP หลักการนี้หมายถึง **Object ที่ต่างชนิดกัน สามารถตอบสนองต่อ "คำสั่ง" (เมธอด) เดียวกัน ในรูปแบบที่เป็นของตัวเองได้**

เพื่อให้เห็นภาพง่ายที่สุด ลองนึกถึงสัตว์ชนิดต่างๆ นะคะ สัตว์ทุกตัว "ส่งเสียงร้อง" ได้ แต่เสียงร้องของมันไม่เหมือนกันถ้าเราสั่งให้ สุนัข "ส่งเสียงร้อง" มันจะ "โฮ่งๆ"ถ้าเราสั่งให้ แมว "ส่งเสียงร้อง" มันจะ "เหมียว"ถ้าเราสั่งให้ วัว "ส่งเสียงร้อง" มันจะ "มอร์~"เห็นมั้ยคะว่า "คำสั่ง" คืออย่างเดียวกัน (ส่งเสียงร้อง) แต่ "ผลลัพธ์" ที่ได้จะแตกต่างกันไป ขึ้นอยู่กับว่าเราสั่ง Object ชนิดไหน นี่แหละค่ะคือหัวใจของ Polymorphism



Polymorphism speak()

```python
class Dog:
    def speak(self):
        print("โฮ่ง! โฮ่ง!")

class Cat:
    def speak(self):
        print("เหมียว~")

class Cow:
    def speak(self):
        print("มอร์~ มอร์~")

# --- สร้าง Object จากแต่ละ Class ---
dog1 = Dog()
cat1 = Cat()
cow1 = Cow()

# --- นำสัตว์ทั้งหมดมารวมกันใน list ---
animals = [dog1, cat1, cow1]

# --- นี่คือ Polymorphism ---
# เราใช้ for loop วนซ้ำและใช้คำสั่ง .speak() เหมือนกันทุกรอบ
for animal in animals:
    animal.speak() # <-- สั่งเหมือนกัน แต่ผลลัพธ์เปลี่ยนไปตามชนิดของ animal
```

result in console:

โฮ่ง! โฮ่ง! เหมียว~ มอร์~ มอร์~

จุดที่น่าทึ่งคือ: โค้ดใน for loop ของเราเรียบง่ายมาก เราแค่สั่ง animal.speak() โดยที่ไม่ต้องไปเช็กเลยว่า animal ในรอบนั้นๆ เป็น Dog, Cat, หรือ Cow มันเป็นหน้าที่ของ Object แต่ละตัวที่จะ "รู้เอง" ว่าเมธอด speak() ของตัวเองต้องทำงานอย่างไร ประโยชน์ของมันคือ เราสามารถเขียนโค้ดที่ยืดหยุ่นและรองรับ Object ใหม่ๆ ในอนาคตได้ง่ายมาก เช่น ถ้าวันหนึ่งเราสร้างคลาส Sheep ที่มีเมธอด speak() ของตัวเอง เราก็แค่โยน Object ของ Sheep เข้าไปใน list animals ได้เลย โดยไม่ต้องแก้ไขโค้ดใน for loop สักนิดเดียว!

สุดยอดไปเลยค่ะ! ตอนนี้เราได้เดินทางผ่าน 4 เสาหลักของ Object-Oriented Programming ครบทั้งหมดแล้วนะคะ 🎉

- **Classes/Objects:** การสร้าง **พิมพ์เขียว** และ **วัตถุ** ที่ใช้งานได้จริง
- **Inheritance:** การ **สืบทอด** คุณสมบัติจากคลาสแม่สู่คลาสลูก
- **Encapsulation:** การ **ห่อหุ้มและปกป้อง** ข้อมูลสำคัญภายใน Object
- **Polymorphism:** การที่ Object ต่างชนิดกัน **ตอบสนองต่อคำสั่งเดียวกัน** ในรูปแบบของตัวเอง

หลักการทั้ง 4 ข้อนี้ทำงานร่วมกันเพื่อช่วยให้เราเขียนโปรแกรมขนาดใหญ่ที่ซับซ้อนได้อย่างเป็นระเบียบ แก้ไขง่าย และนำโค้ดกลับมาใช้ซ้ำได้อย่างมีประสิทธิภาพค่ะ!