# OOP ฉบับ Java

อุษา สัมมาพันธ์

สิงหาคม 2563

# คำนำ

หนังสือเล่มนี้เขียนขึ้นสำหรับผู้ที่เข้าใจการโปรแกรมพื้นฐาน เช่น เงื่อนไข ลูป เมทอด และต้องการเรียนรู้การโปรแกรมเชิงวัตถุ ซึ่งเป็นแนวทางการโปรแกรมที่ต่างจากการโปรแกรมเชิงโครงสร้าง มีการจัดแบ่งรหัสคำสั่งออกเป็นส่วน ๆ ทั้งการเก็บข้อมูลและการประมวลผลที่แตกต่างไป โดยการโปรแกรมเชิงวัตถุจะแบ่งรหัสคำสั่งออกเป็นคลาส ภายในคลาสประกอบด้วยข้อมูลและเมทอด และนำคลาสไปใช้โดยการสร้างอ๊อบเจ็กต์จากคลาส แล้วให้อ๊อบเจ็กต์ทำงานร่วมกันเพื่อประมวลผลตามวัตถุประสงค์ที่ต้องการ อ๊อบเจ็กต์แปลเป็นภาษาไทยว่าวัตถุ ดังนั้น จึงเรียกการโปรแกรมที่ใช้วัตถุในการประมวลผลนี้ว่า การโปรแกรมเชิงวัตถุ

ภาษาโปรแกรมเชิงวัตถุในปัจจุบันมีหลากหลายภาษาให้เลือกใช้ เช่น C++, C#, Java, Smalltalk เป็นต้น หนังสือเล่มนี้จะแนะนำการเขียนโปรแกรมเชิงวัตถุด้วยภาษา Java ซึ่งมีการโปรแกรมพื้นฐาน เช่น เงื่อนไข ลูป คล้ายคลึงกับภาษาโปรแกรมหลายภาษา จึงน่าจะทำให้ผู้ที่มีพื้นฐานภาษาโปรแกรมใด ๆ สามารถเข้าใจหนังสือเล่มนี้ได้ไม่ยาก และสามารถเปรียบเทียบกับภาษานั้น ๆ เพื่อให้เห็นความแตกต่างระหว่างการเขียนโปรแกรมเชิงโครงสร้างและการเขียนโปรแกรมเชิงวัตถุได้ง่าย

หนังสือการโปรแกรมเชิงวัตถุส่วนใหญ่จะอธิบายตั้งแต่การโปรแกรมพื้นฐาน ก่อนจะอธิบายคลาสและอ๊อบเจ็กต์ รวมถึงการสืบทอด แต่ไม่ได้อธิบายการนำคลาสมาประกอบกันโดยละเอียด ทำให้ผู้อ่านยังไม่สามารถนำคลาสมาประกอบกันเพื่อสร้างเป็นโปรแกรมที่ใหญ่และสมบูรณ์ขึ้น

หนังสือเล่มนี้จะเน้นอธิบายแนวทางการโปรแกรมเชิงวัตถุให้ละเอียดและลึกซึ้งขึ้น โดยไม่อธิบายการโปรแกรมพื้นฐาน เนื้อหาจะเริ่มจากคลาสและอ๊อบเจ็กต์ การประกอบกัน การสืบทอด แล้วจึงอธิบายหลักการเชิงวัตถุ เช่น abstraction, encapsulation ให้เข้าใจอย่างลึกซึ้ง พร้อมทั้งแนะนำแนวทางการเขียนโปรแกรมที่ดี มีตัวอย่างให้เรียนรู้และให้เข้าใจง่ายขึ้น ผู้เขียนหวังว่าผู้อ่านจะเข้าใจการเขียนโปรแกรมเชิงวัตถุได้ดีขึ้น สามารถเขียนโปรแกรมเพื่อพัฒนาซอฟต์แวร์ที่มีคุณภาพดีได้ในอนาคต

อุษา สัมมาพันธ์
ภาควิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ มหาวิทยาลัยเกษตรศาสตร์
สิงหาคม 2563

# สารบัญ

# บทที่ 1

# คลาสและอ๊อบเจ็กต์

หลังจากบทที่ 1 ได้แสดงการสร้างคลาสและอ๊อบเจ็กต์เบื้องต้นในภาษาจาวา บทที่ 2 นี้จะอธิบายหลักการและแนวคิดเกี่ยวกับคลาสและอ๊อบเจ็กต์ในการโปรแกรมเชิงวัตถุให้ละเอียดขึ้น การใช้ภาษาจาวาในการสร้างคลาสและอ๊อบเจ็กต์ที่ซับซ้อนขึ้น โดยมีตัวอย่างที่หลากหลายประกอบการอธิบายเพื่อให้เข้าใจคลาสและอ๊อบเจ็กต์ได้ง่าย รวมถึงอธิบายเกร็ดเล็กเกร็ดน้อยเกี่ยวกับคลาสและอ๊อบเจ็กต์

เนื่องจากคุณภาพของซอฟต์แวร์จะแทรกซึมอยู่ในทุกอณูของโปรแกรม ดังนั้น เราควรใส่ใจกับการเขียนโปรแกรม ไม่ว่าจะเป็นส่วนเล็กส่วนน้อย ตั้งแต่การตั้งชื่อ การควบคุมการเข้าถึงส่วนต่าง ๆ การจัดแบ่งคลาส และการเลือกรหัสคำสั่งมาใส่ในเมทอด บทนี้และบทถัด ๆ ไปจึงอธิบายหลักการเขียนคลาสและเมทอด เพื่อให้โปรแกรมมีคุณลักษณะที่ดีด้วย

## 1.1 รู้จักคลาสและอ๊อบเจ็กต์

หัวใจของการเขียนโปรแกรมเชิงวัตถุคือคลาสและอ๊อบเจ็กต์ คลาส (class) เป็นกลไกในภาษาโปรแกรมเชิงวัตถุที่ช่วยจัดระเบียบข้อมูลและรหัสคำสั่งให้เป็นส่วน ๆ โดยรวบรวมตัวแปรข้อมูลและเมทอดที่บรรจุรหัสคำสั่งการประมวลผลที่เกี่ยวข้องกัน มาห่อหุ้มไว้ด้วยกันภายในคลาสเดียวกัน และเมื่อต้องการนำคลาสไปใช้ เราต้องสร้าง “อ๊อบเจ็กต์” (object) ของคลาสขึ้นมา จึงจะสามารถเข้าถึงและนำตัวแปรข้อมูลกับเมทอดภายในคลาสไปใช้ได้

คลาสเป็นเหมือนแม่พิมพ์หรือต้นแบบที่ใช้ในการผลิตอ๊อบเจ็กต์ออกมา อ๊อบเจ็กต์ที่ถูกผลิตขึ้นมานี้จะมีหลากหลายรูปแบบขึ้นอยู่กับคลาสที่เป็นแม่พิมพ์ และอ๊อบเจ็กต์เหล่านี้จะทำงานร่วมกันผ่านการเรียกใช้เมทอดของกันและกัน คำว่า “อ๊อบเจ็กต์” แปลเป็นภาษาไทยว่า วัตถุ จึงเป็นที่มาของการเรียกการโปรแกรม

ในลักษณะนี้ว่า การโปรแกรมเชิงวัตถุ หัวข้อนี้จะมาทำความรู้จักกับคลาสและอ๊อบเจ็กต์ให้ลึกซึ้งขึ้น

## คลาส

ในการจัดระเบียบข้อมูลและรหัสคำสั่งให้เป็นส่วน ๆ เราจะต้องแบ่งโปรแกรมออกเป็นคลาสต่าง ๆ โดยแต่ละคลาสควรมีหน้าที่ที่ชัดเจน โดยทั่วไป เราจะจัดกลุ่มข้อมูลที่มีความสัมพันธ์กัน รวบรวมและห่อหุ้มเอาไว้ด้วยกันภายในคลาสเดียวกัน และมีช่องทางในการเข้าถึงและประมวลผลข้อมูลภายในคลาสนั้น ๆ เพื่อให้คลาสอื่นสามารถเรียกใช้ได้ คลาสจึงประกอบด้วย (1) ข้อมูลและ (2) การประมวลผลข้อมูล โดยข้อมูลจะเก็บในตัวแปรภายในคลาส และการประมวลผลจะเป็นรหัสคำสั่งที่อยู่ภายในเมทอด รหัสคำสั่งในเมทอดนี้จะใช้สำหรับประมวลผลตัวแปรภายในคลาสนั้น ความยาก (และความสนุก) ในการเขียนโปรแกรมเชิงวัตถุคือ การตัดสินใจว่า โปรแกรมของเราจะแบ่งเป็นคลาสใดบ้าง จะกระจายข้อมูลกับเมทอดออกไปไว้ในคลาสเหล่านั้นอย่างไร และจะเชื่อมต่อหรือประกอบคลาสเหล่านั้นเข้าด้วยกันอย่างไร

เราสามารถมองได้ว่า คลาสเป็นการนิยามประเภทข้อมูลขึ้นมาใหม่ (user-defined type) เป็นประเภทข้อมูลที่รวบรวมและห่อหุ้มข้อมูลและเมทอดที่มีความสัมพันธ์กันไว้ด้วยกัน ประเภทข้อมูลที่เป็นคลาสนี้มีความซับซ้อนมากกว่าประเภทข้อมูลพื้นฐาน (primitive type) เนื่องจากคลาสประกอบด้วยข้อมูลประเภทอื่นหลายข้อมูล

การโปรแกรมเพื่อนิยามคลาสสามารถทำได้โดยใช้คีย์เวิร์ด `class` ตามด้วยชื่อคลาสหรือชื่อประเภทข้อมูลใหม่ ภายในคลาสจะประกาศข้อมูลและเมทอดที่เกี่ยวข้องกับคลาสไว้ การตั้งชื่อคลาสควรตั้งให้ตรงกับบริบทการใช้งาน และนิยมตั้งเป็น**คำนาม**เอกพจน์ เนื่องจากคลาสเป็นประเภทข้อมูลที่ใช้เป็นตัวแทนของวัตถุ ตัวอย่างเช่น เราสามารถนิยามคลาส BankAccount ให้เป็นประเภทข้อมูลใหม่ที่รวบรวมและห่อหุ้มข้อมูลกับเมทอดที่เกี่ยวข้องกับบัญชีธนาคาร ประกอบด้วยข้อมูลพื้นฐานคือ ชื่อบัญชี (name) และยอดคงเหลือ (balance) โดยข้อมูลชื่อบัญชีเป็นประเภทสตริงและข้อมูลยอดคงเหลือเป็นประเภทจำนวนจริง ประกอบด้วยเมทอดที่เกี่ยวข้องกับบัญชีธนาคารคือ การฝากเงินเข้าบัญชี (deposit) และการถอนเงินออกจากบัญชี (withdraw) ดังรหัสคำสั่งต่อไปนี้

```
1  class BankAccount {
2
3      String name;
4      double balance;
5
6      void deposit(double amount) {
7          balance += amount;
8      }
9
```

```
10      void withdraw(double amount) {
11          balance -= amount;
12      }
13  }
```

อีกตัวอย่างหนึ่งคือ คลาสนักเรียน (Student) ซึ่งเก็บและประมวลข้อมูลเกี่ยวกับนักเรียน มีข้อมูลชื่อ (name) คะแนนสอบกลางภาค (midtermScore) และคะแนนสอบปลายภาค (finalScore) และมีเมทอดเพื่อคำนวณคะแนนรวม (totalScore) ดังรหัสคำสั่งต่อไปนี้

```
1   class Student {
2
3       String name;
4       int midtermScore, finalScore;
5
6       int totalScore() {
7           return midtermScore + finalScore;
8       }
9   };
```

## อ๊อบเจ็กต์

เมื่อต้องการนำคลาสไปใช้ เราต้องสร้าง “อ๊อบเจ็กต์” หรือวัตถุของคลาสขึ้นมาก่อน เพื่อเข้าถึงและใช้งานข้อมูลในตัวแปรกับเมทอดภายใน โดยอ๊อบเจ็กต์จะเป็นเสมือนข้อมูลที่ผลิตมาจากคลาส และคลาสถือเป็นประเภทของข้อมูลแบบหนึ่ง หากเปรียบเทียบกับข้อมูลประเภทจำนวนเต็ม อาจเปรียบ 5 เป็นอ๊อบเจ็กต์ของคลาสจำนวนเต็มได้

การสร้างอ๊อบเจ็กต์เบื้องต้นจะใช้คีย์เวิร์ด `new` ตามด้วยชื่อคลาส ดังรหัสคำสั่งต่อไปนี้

```
BankAccount annAct = new BankAccount();
BankAccount benAct = new BankAccount();
```

```
Student kwan = new Student();
Student fon = new Student();
```

รหัสคำสั่งในกรอบบนสร้างอ๊อบเจ็กต์หรือวัตถุประเภทบัญชีธนาคาร 2 อ๊อบเจ็กต์ และตั้งชื่ออ๊อบเจ็กต์ทั้งสองนี้ว่า `annAct` และ `benAct` เพื่อแสดงถึงบัญชีธนาคารของแอนและบัญชีธนาคารของเบญ รหัสคำสั่งในกรอบล่างสร้างอ๊อบเจ็กต์ประเภทนักเรียน 2 อ๊อบเจ็กต์ และตั้งชื่ออ๊อบเจ็กต์ทั้งสองนี้ว่า `kwan` และ `fon` เพื่อแสดงถึงนักเรียนสองคนที่ชื่อขวัญและฝนตามลำดับ สังเกตว่า อ๊อบเจ็กต์ที่ถูกสร้างขึ้นมานั้นยังไม่

ได้รับการกำหนดค่าให้กับตัวแปรภายใน เช่น ชื่อบัญชี ชื่อนักเรียน หัวข้อถัดไปจะอธิบายการกำหนดค่าให้กับตัวแปรภายในอ๊อบเจ็กต์นี้ สำหรับภาษาจาวา หากไม่ได้กำหนดค่าให้กับตัวแปรเหล่านี้ จาวาจะให้เป็นค่าเริ่มต้นหรือค่าโดยปริยาย (default value) ตามประเภทข้อมูล เช่น สตริงจะมีค่าโดยปริยายเป็น null จำนวนเต็มและจำนวนทศนิยมจะมีค่าโดยปริยายเป็น 0 และ 0.0 ตามลำดับ

ในชีวิตประจำวันของเรา เมื่อเอ่ยถึงวัตถุ เราจะนึกถึงวัตถุที่เป็นรูปธรรมจับต้องได้ เช่น นักเรียน สินค้า รถยนต์ ในการโปรแกรมเชิงวัตถุนั้น อ๊อบเจ็กต์หรือวัตถุจะเป็นได้ทั้งวัตถุที่เป็นรูปธรรมและวัตถุในเชิงนามธรรม วัตถุในเชิงนามธรรม เช่น บัญชีธนาคาร ความคิดเห็น เป็นต้น ในบางครั้งเราเรียกอ๊อบเจ็กต์ว่า อินสแทนซ์ (instance)

หากจะแยกแยะระหว่างคลาสและอ๊อบเจ็กต์ เราสามารถตรวจสอบได้โดยนำคำนามที่น่าจะเป็นคลาสและคำนามที่น่าจะเป็นอ๊อบเจ็กต์มาวางต่อกันด้วยคำกริยา “เป็น” ในรูปแบบ “อ๊อบเจ็กต์” เป็น “คลาส” และเมื่อวางต่อกันแล้วฟังดูเป็นประโยคที่เข้าใจได้ จะถือว่าประธานของประโยคเป็นอ๊อบเจ็กต์ และกรรมของประโยคเป็นคลาส เช่น ขวัญเป็นนักเรียน ฟังแล้วเข้าใจได้ ดังนั้น ขวัญเป็นอ๊อบเจ็กต์ของคลาสนักเรียน

## 1.2 ข้อมูลและเมทอดของอ๊อบเจ็กต์

คลาสช่วยจัดระเบียบและห่อหุ้มข้อมูลกับรหัสคำสั่งให้เป็นส่วน ๆ คลาสจึงประกอบด้วย (1) ตัวแปรที่เก็บข้อมูลและ (2) เมทอดที่บรรจุรหัสคำสั่งสำหรับประมวลผล ข้อมูลที่รวบรวมมาไว้ในคลาสจะเป็นข้อมูลที่แสดงถึงคุณลักษณะของอ๊อบเจ็กต์ของคลาสนั้น ๆ สำหรับเมทอดที่อยู่ภายในคลาสอาจมองได้ว่าเป็นพฤติกรรมของอ๊อบเจ็กต์ ดังนั้น คลาสหรืออ๊อบเจ็กต์ประกอบด้วย 2 องค์ประกอบหลัก ดังนี้

1. **ข้อมูลของอ๊อบเจ็กต์** หรือคุณลักษณะของอ๊อบเจ็กต์ โดยข้อมูลหรือคุณลักษณะนี้อาจเป็นอัตลักษณ์ของอ๊อบเจ็กต์ เช่น หมายเลขบัญชีธนาคาร ชื่อนักเรียน สีของรถ หรือเป็นสถานะของอ๊อบเจ็กต์ เช่น ยอดคงเหลือในบัญชีธนาคาร คะแนนของนักเรียน เป็นต้น เราเรียกข้อมูลของอ๊อบเจ็กต์นี้ว่า ***ตัวแปรประจำอ๊อบเจ็กต์*** (instance variable)
2. **เมทอดของอ๊อบเจ็กต์** หรือพฤติกรรมของอ๊อบเจ็กต์ จะเป็นการกระทำ การทำงาน หรือหน้าที่ที่อ๊อบเจ็กต์นั้นสามารถทำได้หรือให้ผู้อื่นใช้งานได้ เช่น ฝากหรือถอนบัญชีธนาคารได้ ยืมและคืนหนังสือได้ ขับรถได้ เป็นต้น เราเรียกเมทอดภายในอ๊อบเจ็กต์นี้ว่า ***เมทอดประจำอ๊อบเจ็กต์*** (instance method)

คลาสยังประกอบด้วยองค์ประกอบอื่น เช่น ตัวสร้างอ๊อบเจ็กต์ การควบคุมการเข้าถึงส่วนประกอบต่าง ๆ ภายในคลาส หัวข้อ 1.2 นี้จะอธิบายรายละเอียดของตัวแปรประจำอ๊อบเจ็กต์และเมทอดประจำอ๊อบ

เจ็กต์ หัวข้อ 1.3 จะอธิบายตัวสร้างอ๊อบเจ็กต์ และในบทถัดไปจะอธิบายการควบคุมการเข้าถึง รวมถึงองค์ประกอบอื่นเพิ่มเติม

## ตัวแปรประจำอ๊อบเจ็กต์

ข้อมูลที่เก็บรวบรวมและห่อหุ้มไว้ภายในคลาสเป็นคุณลักษณะที่เกี่ยวข้องกับคลาสหรืออ๊อบเจ็กต์ ถือว่าเป็นอัตลักษณ์หรือสถานะของอ๊อบเจ็กต์ของคลาสนั้น ๆ ในภาษาจาวา ข้อมูลของอ๊อบเจ็กต์ที่ประกาศไว้ภายในคลาสนี้เรียกว่า ***ตัวแปรประจำอ๊อบเจ็กต์*** (instance variable) จากตัวอย่างคลาสบัญชีธนาคาร จะเห็นว่าชื่อบัญชีและยอดคงเหลือในบัญชีเป็นคุณลักษณะของบัญชีธนาคาร เราจึงประกาศตัวแปรทั้งสองนี้ให้เป็นตัวแปรประจำอ๊อบเจ็กต์ของคลาสบัญชีธนาคาร ดังรหัสคำสั่งต่อไปนี้

```
class BankAccount {

    String name;
    double balance;

    // ..... methods .....
};
```

สำหรับตัวอย่างคลาสนักเรียนจะเห็นว่า ชื่อและคะแนนเป็นคุณลักษณะของนักเรียน ดังรหัสคำสั่งต่อไปนี้

```
class Student {

    String name;
    int midtermScore, finalScore;

    // ..... methods .....
};
```

สำหรับคลาสอื่น เช่น คลาสสินค้าอาจประกอบด้วย ชื่อสินค้า ราคาสินค้า จำนวนสินค้าในสต๊อค เป็นต้น

```
class Product {

    String name;
    double price;
    int quantity;

    // ..... methods .....
};
```

เมื่อเรานิยามคลาสและสร้างอ๊อบเจ็กต์แล้ว เราสามารถกำหนดค่าและเรียกใช้ตัวแปรประจำอ๊อบเจ็กต์ในอ๊อบเจ็กต์ที่ต้องการได้ โดยเรียกจากชื่อตัวแปรอ๊อบเจ็กต์ในรูปแบบ

*ชื่ออ๊อบเจ็กต์ . ชื่อตัวแปรประจำอ๊อบเจ็กต์*

เช่น `kwan.name`, `fon.name` รหัสคำสั่งด้านล่างกำหนดค่าให้กับตัวแปรประจำอ๊อบเจ็กต์ของอ๊อบเจ็กต์บัญชีธนาคารและอ๊อบเจ็กต์นักเรียน

```
BankAccount annAct = new BankAccount();
annAct.name = "Ann";
annAct.balance = 1000;

BankAccount benAct = new BankAccount();
benAct.name = "Ben";
benAct.balance = 300;
```

```
Student kwan, fon;

kwan.name = "Kwan";
kwan.midtermScore = 9;
kwan.finalScore = 7;

fon.name = "Fon";
fon.midtermScore = 8;
fon.finalScore = 10;
```

หากสังเกตตัวอย่างรหัสคำสั่งการสร้างอ๊อบเจ็กต์และการกำหนดค่าตัวแปรด้านบน จะเห็นว่า แต่ละอ๊อบเจ็กต์มีค่าตัวแปรประจำอ๊อบเจ็กต์ที่ต่างกันไป ดังนั้น ข้อมูลภายในคลาสเหล่านี้จะมีค่าแปรผันไปตามอ๊อบเจ็กต์ กล่าวคือ ต่างอ๊อบเจ็กต์ ถึงแม้จะเป็นคลาสเดียวกัน จะมีค่าของตัวแปรที่แตกต่างกัน เราจึงเรียกตัวแปรเหล่านี้ว่า ***ตัวแปรประจำอ๊อบเจ็กต์*** (instance variable) หมายถึงข้อมูลที่เป็นสมาชิกของอ๊อบเจ็กต์ เราสามารถมีตัวแปรประจำคลาส (class variable) ได้ด้วย ซึ่งตัวแปรประจำคลาสจะเหมือนกันสำหรับทุกอ๊อบเจ็กต์ในคลาสนั้น ดังจะอธิบายในบทถัดไป

ตัวแปรหรือข้อมูลเหล่านี้เป็นได้ทั้งประเภทข้อมูลพื้นฐานหรือประเภทข้อมูลที่เป็นคลาส โดยประเภทข้อมูลพื้นฐานนี้ประกอบด้วย จำนวนเต็ม จำนวนจริง บูลลีน และคาแรกเตอร์ ในบางครั้งอาจหมายถึงสตริงและวันที่ด้วย ตัวแปรที่เป็นประเภทข้อมูลพื้นฐานนี้ส่วนใหญ่จะเป็นข้อมูลที่อธิบายถึงคุณลักษณะหรือสถานะของอ๊อบเจ็กต์ เช่น ชื่อ สี จำนวน ราคา ดังนั้น ตัวแปรประจำอ๊อบเจ็กต์จึงมีชื่อเรียกอื่นอีกด้วย เช่น attribute, property ซึ่งแปลว่าคุณลักษณะหรือคุณสมบัติ

สำหรับตัวแปรที่มีประเภทเป็นคลาสนั้นจะไม่ใช่คุณลักษณะโดยตรง แต่จะถือว่าเป็นส่วนประกอบของอ๊อบเจ็กต์นั้นหรือเป็นสิ่งที่อ๊อบเจ็กต์นั้นมี เราเรียกการเก็บตัวแปรที่มีประเภทเป็นคลาสว่า เป็นการนำคลาสมาประกอบกัน หรือ composition ซึ่งจะอธิบายต่อไปในบทที่ 4 หัวข้อเรื่อง composition

ตัวแปรประจำอ๊อบเจ็กต์ส่วนใหญ่จะถือว่าเป็นข้อมูลที่เป็นการภายในของอ๊อบเจ็กต์ ไม่ควรให้อ๊อบเจ็กต์จากคลาสอื่นเข้าถึงได้โดยตรง หากต้องการเข้าถึงหรือประมวลผลตัวแปรหล่านี้ ควรทำผ่านเมทอด ภาษาโปรแกรมเชิงวัตถุส่วนใหญ่ รวมถึงจาวา จึงมีกลไกช่วยควบคุมการเข้าถึงข้อมูลภายในอ๊อบเจ็กต์ได้ โดยใช้คีย์เวิร์ด เช่น `private` ดังจะอธิบายในหัวข้อการควบคุมการเข้าถึงตัวแปรในบทที่ 3

## เมทอดประจำอ๊อบเจ็กต์

ในการจัดแบ่งรหัสคำสั่งในโปรแกรมออกเป็นส่วน ๆ นั้น สิ่งสำคัญคือ การจัดแบ่งรหัสคำสั่งของโปรแกรมและนำมากระจายใส่ไว้ในเมทอดของคลาสต่าง ๆ เพื่อให้คลาสอื่นหรือเมทอดอื่นสามารถเรียกใช้งานได้ การจัดแบ่งรหัสคำสั่งออกเป็นเมทอดนี้ควรจัดให้เมทอดหนึ่ง ๆ ทำหน้าที่อย่างใดอย่างหนึ่งที่ชัดเจน ตรงตามพฤติกรรม การทำงาน หรือหน้าที่ที่อ๊อบเจ็กต์ของคลาสนั้นสามารถทำได้หรือให้ผู้อื่นใช้งานได้ ในภาษาจาวาเมทอดเหล่านี้นี้เรียกว่า ***เมทอดประจำอ๊อบเจ็กต์*** (instance method)

เมทอดประกอบด้วยสองส่วน คือ (1) ส่วนหัวเมทอดที่ระบุลักษณะของแต่ละเมทอด และ (2) ส่วนตัวเมทอดที่บรรจุรหัสคำสั่งเอาไว้

*ส่วนหัวเมทอดประจำอ๊อบเจ็กต์* ประกอบด้วย 3 ส่วนหลัก ดังต่อไปนี้

- *ชื่อเมทอด* ควรเป็นคำกริยาที่แสดงถึงการกระทำ การทำงาน หรือหน้าที่ที่อ๊อบเจ็กต์ของคลาสนั้นสามารถทำได้หรือให้ผู้อื่นใช้งานได้
- *รายการของพารามิเตอร์* ประกอบด้วยประเภทข้อมูลและชื่อพารามิเตอร์ที่เมทอดจะรับเข้ามา หากเมทอดไม่รับพารามิเตอร์ จะไม่จำเป็นต้องใส่รายการของพารามิเตอร์
- *ประเภทของผลลัพธ์* เป็นประเภทข้อมูลที่เมทอดจะคืนค่ากลับมา โดยเมทอดสามารถคืนค่าเป็นข้อมูลประเภทพื้นฐานหรือเป็นอ๊อบเจ็กต์ของคลาสใด ๆ ได้ หากเมทอดไม่คืนค่าใด จะต้องระบุประเภทของผลลัพธ์เป็น void

*ส่วนตัวเมทอดประจำอ๊อบเจ็กต์* จะบรรจุรหัสคำสั่งไว้ภายในปีกกา `{` และ `}` รหัสคำสั่งในเมทอดจะใช้ในการประมวลผลตัวแปรประจำอ๊อบเจ็กต์ โดยอาจทำหน้าที่เพียงคืนค่าหรือกำหนดค่าตัวแปรประจำอ๊อบเจ็กต์ตรงๆ คำนวณค่าตัวแปรประจำอ๊อบเจ็กต์ที่ซับซ้อนขึ้น หรือปรับเปลี่ยนค่าของตัวแปรประจำอ๊อบเจ็กต์ได้

สังเกตตัวอย่างคลาสบัญชีธนาคารในหัวข้อ 2.1 จะเห็นว่าคลาสนี้มีเมทอดประจำอ๊อบเจ็กต์ 2 เมทอด คือ `deposit` และ `withdraw`

```
1  class BankAccount {
2      String name;
3      double balance;
4
5      void deposit(double amount) {
6          balance += amount;
7      }
8      void withdraw(double amount) {
9          balance -= amount;
10     }
11 }
```

เปรียบเสมือนเป็นการกระทำที่เราสามารถทำกับบัญชีธนาคารได้ คือ การฝากเงินเข้าบัญชี (deposit) และการถอนเงินออกจากบัญชี (withdraw) ทั้งสองเมทอดนี้มีชื่อที่สอดคล้องกับการกระทำดังกล่าว รับพารามิเตอร์เป็นจำนวนเงิน (`amount`) ที่ต้องการฝากหรือถอน ทั้งสองเมทอดเป็นการประมวลผลและปรับเปลี่ยนค่าของตัวแปรประจำอ๊อบเจ็กต์ `balance` และไม่มีการคืนค่าผลลัพธ์ใด จึงระบุประเภทของผลลัพธ์เป็น void

เนื่องจากตัวแปรประจำอ๊อบเจ็กต์ `balance` อยู่ภายในขอบเขตของคลาสเดียวกันกับเมทอด `deposit` และ `withdraw` เมทอดทั้งสองจึงสามารถเรียกใช้ตัวแปร `balance` โดยไม่ต้องรับค่านี้ผ่านพารามิเตอร์

สำหรับการเรียกใช้เมทอดประจำอ๊อบเจ็กต์นั้น จะคล้ายคลึงกับการเรียกใช้ตัวแปรประจำอ๊อบเจ็กต์ คือ จะเรียกใช้จากชื่ออ๊อบเจ็กต์ในรูปแบบ

*ชื่ออ๊อบเจ็กต์ . ชื่อเมทอดประจำอ๊อบเจ็กต์*

ตัวอย่างเช่น `annAct.deposit(100)` และ `kwan.totalScore()` รหัสคำสั่งด้านล่างเรียกใช้เมทอดประจำอ๊อบเจ็กต์ของอ๊อบเจ็กต์บัญชีธนาคารและอ๊อบเจ็กต์นักเรียน

```
annAct.deposit(100);
benAct.withdraw(50);
```

```
System.out.println("Kwan's score: " + kwan.totalScore());
System.out.println("Fon's score: " + fon.totalScore());
```

เมื่อนำรหัสคำสั่งทั้งหมดของคลาสบัญชีธนาคารและคลาสนักเรียน ทั้งการนิยามคลาส การสร้างอ๊อบเจ็กต์ การกำหนดค่าตัวแปรประจำอ๊อบเจ็กต์ และการเรียกใช้เมทอดประจำอ๊อบเจ็กต์ มาเขียนโปรแกรมรวมกันและบันทึกในไฟล์ ดังรหัสคำสั่งในรูปที่ 1.1 - 1.2 และ 1.4 - 1.5 เมื่อเราคอมไฟล์และรันโปรแกรมจะได้ผลการทำงานดังรูปที่ 1.3 และ 1.6

```
class BankAccount {

    // ---- instance variables ----
    String name;
    double balance;

    // ---- instance methods ----
    void deposit(double amount) {
        balance += amount;
    }
    void withdraw(double amount) {
        balance -= amount;
    }
}
```

รูปที่ 1.1 คลาสบัญชีธนาคาร บักทึกในไฟล์ชื่อ BankAccount.java

สังเกตการเรียกใช้เมทอดประจำอ๊อบเจ็กต์ จะเห็นว่า การทำงานของเมทอดจะกระทำต่ออ๊อบเจ็กต์ที่ถูกเรียกเท่านั้น การฝากเงินเข้าบัญชี `momAct` ทำให้ยอดคงเหลือของบัญชีของแม่เปลี่ยน แต่ไม่ได้ทำให้ยอดคงเหลือของบัญชีของน้องเปลี่ยน เช่นกัน เมื่อเรียกเมทอด `totalScore()` จากอ๊อบเจ็กต์ `kwan` จะให้ค่าที่ต่างไปจากการเรียกจากอ๊อบเจ็กต์ `fon`

ดังนั้น เมทอดประจำอ๊อบเจ็กต์ภายในคลาสเหล่านี้จะประมวลผลแตกต่างกันไปตามอ๊อบเจ็กต์ กล่าวคือ เมทอดประจำอ๊อบเจ็กต์จะประมวลผลตัวแปรประจำอ๊อบเจ็กต์ในอ๊อบเจ็กต์ของตนเอง ไม่เกี่ยวข้องกับอ๊อบเจ็กต์อื่น เราจึงเรียกเมทอดเหล่านี้ว่า ***เมทอดประจำอ๊อบเจ็กต์*** (instance method) หมายถึงเมทอดที่ประจำอยู่ที่อ๊อบเจ็กต์นั้น ๆ เท่านั้น ใช้ประมวลผลข้อมูลที่เป็นตัวแปรประจำอ๊อบเจ็กต์ นอกจากนั้น เราสามารถมีเมทอดประจำคลาส (class method) ได้ด้วย ดังจะอธิบายในบทถัดไป

**ประเภทของพารามิเตอร์ของเมทอด**

จากการเรียกใช้เมทอดประจำอ๊อบเจ็กต์จะเห็นว่าเมทอดประจำอ๊อบเจ็กต์จะประมวลผลแตกต่างกันไปตามอ๊อบเจ็กต์ เนื่องจากในภาษาโปรแกรมเชิงวัตถุ อ๊อบเจ็กต์ที่เรียกเมทอดประจำอ๊อบเจ็กต์นั้นถือเป็น

```
class BankAccountMain {
    public static void main(String[] args) {
        BankAccount annAct = new BankAccount();
        annAct.name = "Ann";
        annAct.balance = 1000;

        BankAccount benAct = new BankAccount();
        benAct.name = "Ben";
        benAct.balance = 300;

        annAct.deposit(100);
        benAct.withdraw(50);

        System.out.println(annAct.name + "'s balance: " + annAct.balance);
        System.out.println(benAct.name + "'s balance: " + benAct.balance);
    }
}
```

รูปที่ 1.2 คลาสการใช้งานบัญชีธนาคาร บักทึกในไฟล์ชื่อ BankAccountMain.java

```
Ann's balance: 1100
Ben's balance: 250
```

รูปที่ 1.3 ผลการทำงานของคลาส BankAccountMain และ BankAccount

```
class Student {

    // ---- instance variables ----
    String name;
    int midtermScore, finalScore;

    // ---- instance methods ----
    int totalScore() {
        return midtermScore + finalScore;
    }
}
```

รูปที่ 1.4 คลาสนักเรียน บักทึกในไฟล์ชื่อ Student.java

```
class StudentMain {

    public static void main(String[] args) {
        Student kwan, fon;
        kwan.name = "Kwan";
        fon.name = "Fon";

        kwan.midtermScore = 9;
        kwan.finalScore = 7;

        fon.midtermScore = 8;
        fon.finalScore = 10;

        System.out.println("Kwan's score: " + kwan.totalScore());
        System.out.println("Fon's score: " + fon.totalScore());
    }
}
```

รูปที่ 1.5 คลาสการใช้งานนักเรียน บักทึกในไฟล์ชื่อ StudentMain.java

```
Kwan's score: 16
Fon's score: 18
```

รูปที่ 1.6 ผลการทำงานของคลาส StudentMain และ Student

พารามิเตอร์ประเภทหนึ่งของเมทอด เรียกว่า อิมพลิสิทพารามิเตอร์ หมายถึง พารามิเตอร์แฝงหรือพารามิเตอร์โดยนัย กล่าวคือ ถือว่าเป็นพารามิเตอร์แม้ว่าจะไม่ได้อยู่ในรายการพารามิเตอร์ของเมทอด

ในภาษาโปรแกรมเชิงวัตถุ พารามิเตอร์ที่ส่งผ่านไปให้เมทอดประจำอ๊อบเจ็กต์สามารถแบ่งได้เป็น 2 ประเภท ดังนี้

- เอ็กซ์พลิสิทพารามิเตอร์ (explicit parameter) เป็นพารามิเตอร์ที่ระบุอย่างชัดแจ้ง อยู่ภายในรายการพารามิเตอร์ของเมทอดประจำอ๊อบเจ็กต์
- อิมพลิสิทพารามิเตอร์ (implicit parameter) เป็นพารามิเตอร์ที่ไม่ได้ระบุอย่างชัดแจ้ง แต่แฝงมากับการเรียกใช้เมทอดประจำอ๊อบเจ็กต์ ในการโปรแกรมเชิงวัตถุ อิมพลิสิทพารามิเตอร์ คือ อ๊อบเจ็กต์ที่เรียกใช้เมทอดนั่นเอง

พิจารณารหัสคำสั่งการเรียกใช้เมทอดประจำอ๊อบเจ็กต์ของคลาสบัญชีธนาคารด้านล่าง

```
annAct.deposit(100);
benAct.withdraw(50);
```

บรรทัดแรกเรียกใช้เมทอดประจำอ๊อบเจ็กต์การฝากเงินเข้าบัญชีธนาคาร มีการฝากเงิน 100 บาทเข้าอ๊อบเจ็กต์ `annAct` ดังนั้น 100 คือ เอ็กซ์พลิสิทพารามิเตอร์ของเมทอดประจำอ๊อบเจ็กต์ `deposit` ส่วนอ๊อบเจ็กต์ `annAct` คือ อิมพลิสิทพารามิเตอร์ของเมทอดประจำอ๊อบเจ็กต์ ในบรรทัดที่ 2 เมทอดประจำอ๊อบเจ็กต์ `withdraw` มีเอ็กซ์พลิสิทพารามิเตอร์เป็น 50 และอิมพลิสิทพารามิเตอร์เป็นอ๊อบเจ็กต์ `benAct` ดังนั้น เนื่องจากทั้งสองเมทอดประจำอ๊อบเจ็กต์รับอ๊อบเจ็กต์ที่ต่างกันเป็นอิมพลิสิทพารามิเตอร์ การประมวลผลจึงไม่มีผลเกี่ยวข้องกัน แม้จะเป็นเมทอดจากคลาสเดียวกันก็ตาม

ในการโปรแกรมเชิงวัตถุ การรับอ๊อบเจ็กต์เป็นอิมพลิสิทพารามิเตอร์เมื่อเรียกเมทอดประจำอ๊อบเจ็กต์ในลักษณะนี้ ทำให้รหัสคำสั่งมีลักษณะเหมือนประโยคบอกเล่า กล่าวคือ ชื่ออ๊อบเจ็กต์จะเป็นประธานของประโยค ชื่อเมทอดประจำอ๊อบเจ็กต์จะเป็นกริยาของประโยค และเอ็กซ์พลิสิทพารามิเตอร์จะเป็นกรรมของประโยค ส่งผลให้รหัสคำสั่งมีความคล้ายคลึงกับภาษาธรรมชาติมากกว่าการโปรแกรมเชิงโครงสร้าง ที่รับพารามิเตอร์แบบเอ็กซ์พลิสิททั้งหมด ทำให้โปรแกรมอ่านเข้าใจได้ง่ายขึ้น

## 1.3 คอนสตรักเตอร์

คอนสตรักเตอร์ (constructor) หรือตัวสร้างอ๊อบเจ็กต์มีลักษณะคล้ายเมทอดประจำอ๊อบเจ็กต์ ช่วยในการเตรียมอ๊อบเจ็กต์ให้พร้อมใช้งาน โดยช่วยกำหนดค่าเริ่มต้นให้กับตัวแปรประจำอ๊อบเจ็กต์ หรือประมวลผลรหัสคำสั่งใด ๆ เช่น การตรวจสอบช่วงของค่าเริ่มต้น การเรียกใช้เมทอด การเชื่อมต่อส่วนประกอบอื่น ที่จำเป็นขณะสร้างอ๊อบเจ็กต์ คอนสตรักเตอร์จะถูกเรียกขณะสร้างอ๊อบเจ็กต์เพียงหนึ่งครั้งต่อหนึ่งอ๊อบเจ็กต์เท่านั้น การนิยามคอนสตรักเตอร์จะต้องตั้งชื่อเดียวกับชื่อคลาส ไม่มีการประกาศประเภทของผลลัพธ์ และสามารถรับหรือไม่รับค่าผ่านทางพารามิเตอร์ก็ได้ ตัวอย่างรหัสคำสั่งต่อไปนี้เพิ่มคอนสตรักเตอร์เข้าไปในคลาสบัญชีธนาคาร โดยเพิ่มเข้าไปหลังการประกาศตัวแปรประจำอ๊อบเจ็กต์

```
class BankAccount {

    // ---- instance variables ----
    String name;
    double balance;

    // ---- constructor ----
    BankAccount(String n, double b) {
        name = n;
        balance = b;
    }

    // ---- instance methods ----
    void deposit(double amount) {
        balance += amount;
    }
    void withdraw(double amount) {
        balance -= amount;
    }
}
```

ในรหัสคำสั่งข้างต้น คอนสตรักเตอร์ที่เพิ่มเข้ามารับพารามิเตอร์ 2 ค่าคือชื่อบัญชีธนาคาร `n` และยอดคงเหลือ `b` โดยนำค่าทั้งสองมากำหนดค่าให้กับตัวแปรประจำอ๊อบเจ็กต์ `name` และ `balance` ตามลำดับ หลังจากกำหนดให้คลาสมีคอนสตรักเตอร์แล้ว เมื่อสร้างอ๊อบเจ็กต์ด้วยคีย์เวิร์ด `new` และชื่อคลาส จะต้องส่งพารามิเตอร์เข้าไปกับชื่อคลาสด้วย ดังรหัสคำสั่งค่อไปนี้

```
BankAccount annAct = new BankAccount("Ann", 1000);
BankAccount benAct = new BankAccount("Ben", 300);
```

รหัสคำสั่งข้างต้นสร้างอ๊อบเจ็กต์บัญชีธนาคารขึ้นมาสองอ๊อบเจ็กต์ `annAct` และ `benAct` รหัสคำสั่งนี้ต่างจากการสร้างอ๊อบเจ็กต์ในหัวข้อ 1.1 ตรงการส่งพารามิเตอร์ไปกับชื่อคลาสหลังคีย์เวิร์ด `new` ซึ่งจะเป็นการส่งค่าชื่อบัญชีธนาคารและยอดคงเหลือเข้าไปในคอนสตรักเตอร์ เพื่อกำหนดค่าตัวแปรประจำอ๊อบเจ็กต์ภายในอ๊อบเจ็กต์ที่กำลังสร้างนั่นเอง

การกำหนดค่าเริ่มต้นให้กับตัวแปรประจำอ๊อบเจ็กต์ผ่านคอนสตรักเตอร์นี้ จะให้ผลการทำงานตรงกับรหัสคำสั่งการกำหนดค่าในหัวข้อ 1.2 โดยในหัวข้อ 1.2 การกำหนดค่าเริ่มต้นให้กับตัวแปรประจำอ๊อบเจ็กต์จะทำหลังจากการสร้างอ๊อบเจ็กต์แล้ว และจะกำหนดค่าตัวแปรประจำอ๊อบเจ็กต์ผ่านตัวแปรอ๊อบเจ็กต์ เช่น `annAct.name = "Ann"` และ `annAct.balance = 1000` การกำหนดค่าแยกจากการสร้างอ๊อบเจ็กต์ในลักษณะนี้มีข้อเสียคือ หากโปรแกรมเมอร์ไม่ได้กำหนดค่าเริ่มต้นไว้ คอมไพเลอร์ของภาษาจาวาไม่มีการ

เตือน ทำให้โปรแกรมเมอร์มีโอกาสลืมกำหนดค่าที่จำเป็นได้

ในทางตรงกันข้าม หากเราใช้คอนสตรักเตอร์แล้ว ถ้าโปรแกรมเมอร์ไม่ได้มีการกำหนดเริ่มผ่านคอนสตรักเตอร์ที่ระบุไว้ คอมไพเลอร์ของภาษาจาวาจะแจ้งเตือน ดังนั้น การใช้คอนสตรักเตอร์จะช่วยบังคับให้การสร้างอ๊อบเจ็กต์ต้องกำหนดค่าที่จำเป็นผ่านพารามิเตอร์ของคอนสตรักเตอร์ ทำให้โปรแกรมมีความผิดพลาดน้อยลง นอกจากนั้น การกำหนดค่าผ่านคอนสตรักเตอร์ยังทำให้รหัสคำสั่งสั้นลงอีกด้วย

ดังที่กล่าวไว้ข้างต้น คอนสตรักเตอร์จะช่วยในการเตรียมอ๊อบเจ็กต์ให้พร้อมใช้งาน นอกเหนือจากการกำหนดค่าเริ่มต้นทั่วไปแล้ว ยังสามารถมีรหัสคำสั่งอื่น เช่น การตรวจสอบช่วงของค่าเริ่มต้นหรือการเรียกใช้เมทอดประจำอ๊อบเจ็กต์ที่จำเป็นขณะสร้างอ๊อบเจ็กต์ได้ รหัสคำสั่งต่อไปนี้ตรวจสอบความถูกต้องของค่าเริ่มต้นยอดคงเหลือก่อนว่าต้องไม่ใช่ค่าลบ หากเป็นค่าลบจะกำหนดให้ยอดคงเหลือมีค่าเป็น 0 แทน

```
class BankAccount {

    // ---- instance variables ----
    String name;
    double balance;

    // ---- constructor ----
    BankAccount(String n, double b) {
        name = n;
        balance = b;
        if (b < 0)
            balance = 0;
    }

    // ---- instance methods ----
    void deposit(double amount) {
        balance += amount;
    }
    void withdraw(double amount) {
        balance -= amount;
    }
}
```

## 1.4 การควบคุมการเข้าถึง

สำหรับคลาสบางคลาส ข้อมูลที่อยู่ภายในอาจเป็นข้อมูลสำคัญที่ไม่ต้องการเปิดเผยให้ภายนอกเข้าถึงได้ หรือข้อมูลอาจต้องอยู่ในรูปแบบที่เฉพาะเจาะจง ทำให้ต้องการความปลอดภัย เพื่อป้องกันการแก้ไขที่

ผิดรูปแบบ การเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ในบทที่ผ่านมา อนุญาตให้อ๊อบเจ็กต์อื่นสามารถอ่านและเปลี่ยนแปลงค่าได้โดยง่าย เนื่องจากไม่มีการควบคุมการเข้าถึงหรือควบคุมการเปลี่ยนแปลงตัวแปรตามรูปแบบที่กำหนดได้

จากตัวอย่างคลาสบัญชีธนาคาร ข้อมูลยอดคงเหลือถือได้ว่าเป็นข้อมูลที่ต้องการความปลอดภัย หากมีการกำหนดค่ายอดคงเหลือในบัญชีโดยตรงให้เป็นค่าลบดังตัวอย่างรหัสคำสั่งด้านล่าง บัญชีธนาคารจะอยู่ในสถานะที่ไม่ถูกต้อง ดังนั้น ภาษาโปรแกรมเชิงวัตถุส่วนใหญ่จึงมีกลไกให้โปรแกรมเมอร์สามารถควบคุมการเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ รวมถึงการเข้าถึงคอนสตรักเตอร์และเมทอดประจำอ๊อบเจ็กต์ด้วย

```
BankAccount momAct = new BankAccount("Mom", 1000);
momAct.balance = -500
```

ในภาษาจาวา การควบคุมการเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ และเมทอดประจำอ๊อบเจ็กต์ จะทำได้ 4 ระดับ คือ private, public, package และ protected แต่ละระดับจะอนุญาตให้คลาสและเมทอดภายนอกสามารถเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ และเมทอดประจำอ๊อบเจ็กต์ได้ในรูปแบบที่แตกต่างกัน ในการควบคุมการเข้าถึง เราต้องระบุคีย์เวิร์ดที่บ่งบอกการเข้าถึง เรียกว่า ***access specifier*** หรือ ***access modifier*** ให้ตรงกับระดับการเข้าถึงที่ต้องการ โดยจะระบุด้านหน้าการประกาศตัวแปรประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ และเมทอดประจำอ๊อบเจ็กต์ หัวข้อนี้จะอธิบายระดับ private, public และ package ก่อน สำหรับระดับ protected จะกล่าวถึงในบทถัดไป

- ในระดับส่วนตัวหรือไพรเวท (***private***) อ๊อบเจ็กต์และเมทอดของคลาสอื่นใดจะไม่สามารถเข้าถึงได้โดยตรง ต้องเป็นอ๊อบเจ็กต์ของคลาสที่ประกาศตัวแปรประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ และเมทอดประจำอ๊อบเจ็กต์นั้น ๆ เท่านั้น จึงจะเข้าถึงได้ โดยระบุคีย์เวิร์ด `private`
- ในระดับสาธารณะหรือพับบลิก (***public***) อ๊อบเจ็กต์และเมทอดของคลาสอื่นใดสามารถเข้าถึงได้โดยตรง โดยระบุคีย์เวิร์ด `public`
- ในระดับแพคเกจ (***package***) อ๊อบเจ็กต์และเมทอดของคลาสที่อยู่ในโฟลเดอร์หรือ package เดียวกัน สามารถเข้าถึงได้ ระดับแพคเกจนี้จะไม่ต้องระบุคีย์เวิร์ด ถือเป็นระดับปริยาย (default)

สำหรับการนิยามคลาสที่ผ่านมาในหนังสือเล่มนี้ ไม่ได้มีการระบุการควบคุมการเข้าถึงใด ๆ การเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ และเมทอดประจำอ๊อบเจ็กต์ จึงเป็นแบบแพคเกจ ทำให้อ๊อบเจ็กต์ของคลาสอื่น รวมถึง คลาสที่มีเมทอด `main()` ซึ่งอยู่ในโฟลเดอร์เดียวกันสามารถเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ และเมทอดประจำอ๊อบเจ็กต์ได้ทั้งหมด

อย่างไรก็ตาม โดยทั่วไป เราควรกำหนดตัวแปรประจำอ๊อบเจ็กต์ให้อยู่ในระดับ private เพื่อควบคุมไม่ให้อ๊อบเจ็กต์จากคลาสอื่นเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ได้โดยตรง และควรกำหนดคอนสตรักเตอร์และ

เมทอดประจำอ๊อบเจ็กต์ให้อยู่ในระดับ public (มีกรณีพิเศษที่เราอาจประกาศคอนสตรักเตอร์และเมทอดประจำอ๊อบเจ็กต์ให้อยู่ในระดับ private ได้ ซึ่งจะอธิบายในบทถัด ๆ ไป)

เมื่อเราประกาศตัวแปรประจำอ๊อบเจ็กต์ให้เป็น private แล้ว การเข้าถึงตัวแปรประจำอ๊อบเจ็กต์เหล่านี้โดยตรงผ่านคำสั่ง เช่น `momAct.balance = -500;` จะทำไม่ได้ ดังนั้น คลาสจะต้องมีเมทอดประจำอ๊อบเจ็กต์เพื่อเข้าถึงตัวแปรประจำอ๊อบเจ็กต์เหล่านี้แทน การใช้เมทอดประจำอ๊อบเจ็กต์ในการเข้าถึงนั้น จะทำให้คลาสสามารถควบคุมการกำหนด การเปลี่ยนแปลง และการอ่านค่าในรูปแบบที่ต้องการได้ ส่งผลให้ตัวแปรประจำอ๊อบเจ็กต์มีความปลอดภัย

คลาสบัญชีธนาคารด้านล่างกำหนดให้ตัวแปรประจำอ๊อบเจ็กต์ทั้งหมดให้เป็น private ส่วนคอนสตรักเตอร์และเมทอดประจำอ๊อบเจ็กต์จะเป็น public และเพื่อให้ยอดคงเหลือมีค่าที่ถูกต้อง จึงเพิ่มเงื่อนไขเพื่อตรวจสอบพารามิเตอร์ที่จะใช้กำหนดค่า ทั้งในคอนสตรักเตอร์และเมทอดประจำอ๊อบเจ็กต์ `deposit()` และ `withdraw()` ทำให้ค่ายอดคงเหลือไม่สามารถเป็นค่าลบได้เลย

```
1  class BankAccount {
2
3      // ---- instance variables ----
4      private String name;
5      private double balance;
6
7      // ---- constructors ----
8      public BankAccount(String n, double b) {
9          name = n;
10         balance = b;
11         if (b < 0)
12             balance = 0;
13     }
14
15     // ---- instance methods ----
16     public void deposit(double amount) {
17         if (amount > 0)
18             balance += amount;
19     }
20
21     public void withdraw(double amount) {
22         if (amount > 0 && amount < balance)
23             balance -= amount;
24     }
25 }
```

จากรหัสคำสั่งข้างต้น จะเห็นว่าตัวแปรประจำอ๊อบเจ็กต์ถูกประกาศหลังคีย์เวิร์ด `private` เราจึงไม่

สามารถอ่านและกำหนดค่าให้ตัวแปรประจำอ๊อบเจ็กต์โดยตรงได้ พิจารณารหัสคำสั่งต่อไปนี้

```
1 BankAccount momAct = new BankAccount("Mom");
2 momAct.balance = -500;
```

บรรทัดที่ 2 กำหนดค่ายอดคงเหลือ `balance` เป็น -500 โดยตรง ซึ่งเมื่อคอมไพล์โปรแกรมแล้ว จะมีการแจ้งข้อผิดพลาดดังนี้ ทำให้ไม่สามารถกำหนดค่าในลักษณะนี้ได้อีกต่อไป

```
'balance' has private access in 'BankAccount'
```

นอกจากนั้น เราได้เพิ่มการตรวจสอบเงื่อนไขในการกำหนดค่ายอดคงเหลือดังนี้

- คอนสตรักเตอร์มีการตรวจสอบว่า พารามิเตอร์ `b` มีค่าน้อยกว่า 0 หรือไม่ ถ้าน้อยกว่า 0 จะกำหนดค่าให้เป็น 0 เพื่อให้ยอดคงเหลือไม่เป็นค่าลบ
- เมทอดประจำอ๊อบเจ็กต์ `deposit()` จะปรับเปลี่ยนค่ายอดคงเหลือเมื่อจำนวนเงินฝาก (`amount`) เป็นค่าบวกเท่านั้น หากฝากด้วยค่าลบ จะไม่มีการปรับเปลี่ยนค่ายอดคงเหลือ `balance`
- เมทอดประจำอ๊อบเจ็กต์ `withdraw()` เช่นกัน จะปรับเปลี่ยนค่ายอดคงเหลือเมื่อจำนวนเงินถอน (`amount`) เป็นค่าบวกเท่านั้น หากถอนด้วยค่าลบ จะไม่มีการปรับเปลี่ยนค่ายอดคงเหลือ `balance` นอกจากนั้น ยังตรวจสอบด้วยว่า จะถอนได้ก็ต่อเมื่อจำนวนที่ต้องการถอนนั้น น้อยกว่ายอดคงเหลือที่มี

เนื่องจากการประกาศตัวแปรประจำอ๊อบเจ็กต์ให้เป็น private จะทำให้ไม่สามารถอ่านค่าตัวแปรประจำอ๊อบเจ็กต์เหล่านี้ เพื่อนำไปแสดงผลหรือคำนวณต่อได้ เราจึงต้องสร้างเมทอดประจำอ๊อบเจ็กต์เพิ่มเติม เพื่อช่วยอ่านและกำหนดค่าตัวแปรประจำอ๊อบเจ็กต์ โดยเรานิยมใช้เมทอดประจำอ๊อบเจ็กต์ประเภท “getter” และ “setter” ซึ่งจะอธิบายในหัวข้อถัดไป

## 1.5 เมทอดประจำอ๊อบเจ็กต์ประเภท getter และ setter

เมทอดประจำอ๊อบเจ็กต์ประเภท “getter” และ “setter” ใช้ในการอ่านค่าและกำหนดค่าให้กับตัวแปรประจำอ๊อบเจ็กต์ที่เป็น private ส่วนใหญ่เมทอดประเภท getter จะทำหน้าที่เพียงคืนค่าตัวแปรประจำอ๊อบเจ็กต์ตรงๆ เท่านั้น ส่วนเมทอดประเภท setter จะใช้ในการกำหนดค่า โดยอาจกำหนดค่าตรง ๆ จากค่าที่รับเข้ามาผ่านพารามิเตอร์ หรืออาจตรวจสอบเงื่อนไขก่อนที่จะกำหนดค่า เรานิยมตั้งชื่อเมทอดประจำ

อ๊อบเจ็กต์หล่านี้โดยขึ้นต้นด้วยคำว่า get หรือ set ต่อด้วยชื่อตัวแปรประจำอ๊อบเจ็กต์ สำหรับการอ่านค่าและกำหนดค่าตามลำดับ เช่น getName() หรือ setName() เป็นต้น

เราไม่จำเป็นต้องมีเมทอด “getter” และ “setter” สำหรับทุกตัวแปรประจำอ๊อบเจ็กต์ จะมีก็ต่อเมื่อจำเป็นต้องใช้เท่านั้น เช่น หากคลาสนักเรียนมีตัวแปรประจำอ๊อบเจ็กต์ ชื่อ และรหัสนักเรียน โดยทั่วไปนักเรียนสามารถขอเปลี่ยนชื่อได้ แต่จะไม่สามารถขอเปลี่ยนรหัสนักเรียนได้ ดังนั้น เมื่อกำหนดค่าเริ่มต้นให้กับชื่อและรหัสนักเรียนแล้ว ไม่ควรเปลี่ยนรหัสนักเรียนได้ จึงไม่ควรมี setter สำหรับรหัสนักเรียน รหัสคำสั่งคลาสบัญชีธนาคารต่อไปนี้เพิ่มเมทอดประจำอ๊อบเจ็กต์ประเภท getter และ setter ที่เหมาะสม

```
1  class BankAccount {
2
3      // ---- instance variables ----
4      private String name;
5      private double balance;
6
7      // ---- constructors ----
8      public BankAccount(String n, double b) {
9          name = n;
10         balance = b;
11         if (b < 0)
12             balance = 0;
13     }
14
15     // ---- instance methods ----
16     public void deposit(double amount) {
17         if (amount > 0)
18             balance += amount;
19     }
20
21     public void withdraw(double amount) {
22         if (amount > 0 && amount < balance)
23             balance -= amount;
24     }
25
26     // ---- getters and setters ---
27     public String getName() {
28         return name;
29     }
30
31     public double getBalance() {
32         return balance;
33     }
34
```

```
35      public void setName(String n) {
36          name = n;
37      }
38  }
```

รหัสคำสั่งข้างต้นในบรรทัดที่ 34 - 44 ได้เพิ่มเมทอด getter สำหรับตัวแปรประจำอ๊อบเจ็กต์ `name` และ `balance` แต่เพิ่มเมทอด setter สำหรับตัวแปรประจำอ๊อบเจ็กต์ `name` เท่านั้น เนื่องจากเราสามารถขอเปลี่ยนชื่อบัญชีได้ หากมีการเปลี่ยนชื่อโดยถูกต้องตามกฎหมายแล้ว แต่เราไม่ควรกำหนดค่าให้กับยอดคงเหลือ `balance` ได้ เพราะว่าคลาสบัญชีธนาคารมีเมทอด `deposit()` และ `withdraw()` ที่ช่วยกำหนดค่ายอดคงเหลือให้ถูกต้องอยู่แล้ว เมื่อคลาสบัญชีธนาคารมีเมทอด getter และ setter แล้ว เราจะสามารถอ่านและกำหนดค่าให้กับตัวแปรประจำอ๊อบเจ็กต์ที่เป็น private ได้ ดังรหัสคำสั่งต่อไปนี้

```
1  BankAccount namAct = new BankAccount("Nam", -50);
2  System.out.println("Nam: " + namAct.getBalance());
3
4  BankAccount fahAct = new BankAccount("Fah", 100);
5  fahAct.deposit(-10);
6  fahAct.withdraw(800);
7
8  System.out.println("Fah: " + fahAct.getBalance());
```

บรรทัดที่ 1 พยายามสร้างอ๊อบเจ็กต์เพื่อให้ยอดคงเหลือมีค่าเป็น -50 แต่คอนสตรักเตอร์มีเงื่อนไขว่าหากพารามิเตอร์ยอดคงเหลือมีค่าเป็นลบ จะกำหนดยอดคงเหลือให้เป็น 0 ดังนั้น เมื่อบรรทัดที่ 2 พิมพ์ค่ายอดคงเหลือออกมา จึงได้ค่าเป็น 0 ดังผลการรันด้านล่าง นอกจากนั้น บรรทัดที่ 5 มีการฝากเงินด้วยจำนวนลบ และบรรทัดที่ 6 พยายามถอนเงินด้วยจำนวนเงินที่มากกว่ายอดคงเหลือ ซึ่งทั้งสองบรรทัดนี้จะไม่สามารถเปลี่ยนยอดคงเหลือได้ เนื่องจากทั้งเมทอด `deposit()` และ `withdraw()` มีการตรวจสอบเงื่อนไขไว้เพื่อไม่ให้ค่ายอดคงเหลืออยู่ในสถานะที่ไม่ถูกต้อง ทำให้การพิมพ์ค่าในบรรทัดที่ 7 ยังคงมีค่ายอดคงเหลือเหมือนกับการสร้างอ๊อบเจ็กต์ในบรรทัดที่ 4

```
Nam: 0
Fah: 100
```

## 1.6 เมทอด toString()

เมื่อเราประกาศตัวแปรประจำอ๊อบเจ็กต์ให้เป็น private และนิยามเมทอดประจำอ๊อบเจ็กต์ประเภท “getter” และ “setter” ในการเข้าถึงแล้ว การพิมพ์ค่าตัวแปรประจำอ๊อบเจ็กต์ทั้งหมดออกมาจะทำให้รหัสคำสั่งยาวเกินจำเป็น ดังนี้

```
BankAccount fahAct = new BankAccount("Fah", 100);
System.out.println(fahAct.getName() + " has balance " +
fahAct.getBalance());
```

เราจึงนิยมเขียนเมทอดประจำอ๊อบเจ็กต์ `toString()` เพื่อคืนค่าสตริงในรูปแบบที่กำหนด ช่วยให้การพิมพ์ค่าของอ๊อบเจ็กต์ทำได้อย่างกระชับ รหัสคำสั่งด้านล่างแสดงตัวอย่างเมทอดประจำอ๊อบเจ็กต์ `toString()` นี้ โดยคืนค่าเป็นสตริงในรูปแบบที่ต้องการพิมพ์ค่า

```
class BankAccount {

    //.... instance variables, constructors, instance methods ....

    public String toString() {
        return name + " has balance " + balance;
    }
}
```

เมื่อประกาศเมทอดประจำอ๊อบเจ็กต์ `toString()` แล้ว เราสามารถพิมพ์ค่าของอ๊อบเจ็กต์ได้ดังรหัสคำสั่งด้านล่าง โดยใส่อ๊อบเจ็กต์เข้าไปในเมทอด `System.out.println` ได้เลย เนื่องจากเมทอดนี้จะเรียกเมทอด `toString()` ของอ๊อบเจ็กต์ให้โดยอัตโนมัติ จะเห็นว่า โปรแกรมกระชับขึ้นและอ่านเข้าใจได้ง่ายขึ้น

```
BankAccount fahAct = new BankAccount("Fah", 100);
System.out.println(fahAct);
```

## 1.7 พอยน์เตอร์ this

จากหัวข้อที่ผ่านมา สังเกตการกำหนดค่าให้กับตัวแปรประจำอ๊อบเจ็กต์ ทั้งในคอนสตรักเตอร์และในเมทอดประจำอ๊อบเจ็กต์ setter ที่มีการรับค่าเข้ามาผ่านพารามิเตอร์ เราตั้งชื่อพารามิเตอร์เหล่านี้โดยใช้ตัวอักษรแรกของชื่อตัวแปรประจำอ๊อบเจ็กต์ เช่น คอนสตรักเตอร์ของคลาสบัญชีธนาคารต่อไปนี้

```
public BankAccount(String n, double b) {
    name = n;
    balance = b;
}
```

และเมทอดประจำอ๊อบเจ็กต์ที่เป็น setter ของคลาสบัญชีธนาคารต่อไปนี้

```
public void setName(String n) {
    name = n;
}
```

การตั้งชื่อด้วยอักษรย่อแบบนี้ ไม่ค่อยสื่อความหมาย หากสามารถใช้ชื่อเต็มได้ น่าจะสื่อความหมายมากกว่าและช่วยให้เข้าใจรหัสคำสั่งได้ง่ายขึ้น อย่างไรก็ตาม หากเราใช้ชื่อเต็มดังตัวอย่างรหัสคำสั่งด้านล่าง ชื่อพารามิเตอร์จะไปซ้ำกับตัวแปรประจำอ๊อบเจ็กต์

```
public BankAccount(string name, double balance) {
    name = name;
    balance = balance;
}
```

เนื่องจากในจาวาเมื่อมีการประกาศชื่อซ้ำในขอบเขตที่ซ้อนกันอยู่ (ในที่นี้คือ ตัวแปร `name` ซึ่งมีการประกาศเป็นตัวแปรประจำอ๊อบเจ็กต์ในขอบเขตคลาสและประกาศเป็นพารามิเตอร์ในขอบเขตคอนสตรักเตอร์ที่ซ้อนอยู่ในคลาส) ชื่อตัวแปรนั้นจะหมายถึงตัวแปรที่ได้รับการประกาศที่ขอบเขตล่าสุดหรือใกล้ที่สุด โดยในที่นี้ตัวแปร `name` ที่ได้รับการประกาศล่าสุด คือ พารามิเตอร์ของคอนสตรักเตอร์ ดังนั้น จาวาจึงมองว่า ตัวแปร `name` ทั้งทางซ้ายมือและขวามือของเครื่องหมาย = ในรหัสคำสั่งด้านบนเป็นพารามิเตอร์ทั้งคู่ ส่งผลให้ตัวแปรประจำอ๊อบเจ็กต์ไม่ได้รับการกำหนดค่า

จาวามีคีย์เวิร์ด `this` ที่จะช่วยแยกแยะระหว่างตัวแปรประจำอ๊อบเจ็กต์และพารามิเตอร์เมื่อทั้งสองตัวแปรมีชื่อเดียวกัน โดย `this` เป็นตัวอ้างอิง (reference) พิเศษที่ใช้ภายในคลาสในการอ้างอิงไปถึงอ๊อบเจ็กต์ของคลาสนี้ ทำให้สามารถเรียกใช้ตัวแปรและเมทอดประจำอ๊อบเจ็กต์ของคลาสนี้ผ่าน `this` ได้ เมื่อใช้ `this` กับตัวแปร จะทำให้จาวารู้ว่าตัวแปรนี้เป็นตัวแปรประจำอ๊อบเจ็กต์ ไม่ใช่พารามิเตอร์

รหัสคำสั่งด้านล่างแสดงการใช้งานตัวอ้างอิง `this` โดย `this.name` จะหมายถึงตัวแปรประจำอ๊อบเจ็กต์ ส่วน `name` จะหมายถึงพารามิเตอร์ ทำให้การกำหนดค่าให้ตัวแปรประจำอ๊อบเจ็กต์โดยใช้ค่าจากพารามิเตอร์ทำงานได้ถูกต้อง โดยชื่อพารามิเตอร์จะสื่อความหมายมากกว่าชื่อที่เป็นอักษรย่อ

```
public BankAccount(String name, double balance) {
    this.name = name;
    this.balance = balance;
}
```

```
public void setName(String name) {
    this.name = name;
}
```

ตัวอ้างอิง `this` นี้ยังมีประโยชน์อื่นด้วย เช่น เมื่อต้องการคืนค่าเป็นอ๊อบเจ็กต์ของคลาสนี้ สามารถคืนค่าโดยใช้ตัวอ้างอิง `this` ได้ ดังจะเห็นการใช้งานในบทถัด ๆ ไป

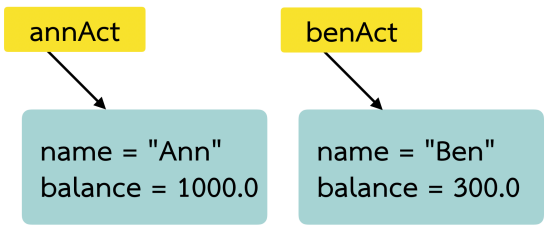
## 1.8 ทำความเข้าใจกับอ๊อบเจ็กต์

เพื่อให้เข้าใจอ๊อบเจ็กต์ในการโปรแกรมเชิงวัตถุ การวาดรูปอ๊อบเจ็กต์ที่ถูกสร้างขึ้นมาจะช่วยให้จินตนาการเกี่ยวกับอ๊อบเจ็กต์ได้ง่ายขึ้น รูปอ๊อบเจ็กต์ในหัวข้อนี้ไม่ใช่วิธีมาตรฐานในการวาดอ๊อบเจ็กต์ แต่เป็นการวาดเพื่อให้เข้าใจการคงอยู่ของอ๊อบเจ็กต์ในโปรแกรมเท่านั้น

รหัสคำสั่งต่อไปนี้เป็นการสร้างอ๊อบเจ็กต์ของคลาสบัญชีธนาคารขึ้นมาสองอ๊อบเจ็กต์

```
BankAccount annAct = new BankAccount("Ann", 1000);
BankAccount benAct = new BankAccount("Ben", 300);
```

เปรียบเสมือนการสร้างกล่องอ๊อบเจ็กต์ขึ้นมาสองกล่อง มีชื่อ `annAct` และ `benAct` ตามชื่ออ๊อบเจ็กต์ที่เราสร้างขึ้น ดังรูปที่ 1.7 แต่ละกล่องจะประกอบด้วยตัวแปรประจำอ๊อบเจ็กต์ `name` และ `balance` ในคลาสบัญชีธนาคาร

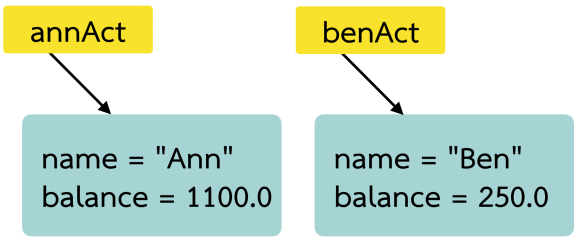



รูปที่ 1.7 อ๊อบเจ็กต์ของคลาบัญชีธนาคารจำนวนสองอ๊อบเจ็กต์

เมื่อเราเรียกใช้เมทอดประจำอ๊อบเจ็กต์จากอ๊อบเจ็กต์ annAct และ benAct ตามรหัสคำสั่งต่อไปนี้

```
annAct.deposit(100);
benAct.withdraw(50);
```

จะเกิดการประมวลผลตัวแปรประจำอ๊อบเจ็กต์ที่เป็นของอ๊อบเจ็กต์นั้น ๆ เท่านั้น ทำให้ได้ภาพดังรูปที่ 1.8 ต่อไปนี้




รูปที่ 1.8 หลังการเรียกเมทอดประจำอ๊อบเจ็กต์จากอ๊อบเจ็กต์บัญชีธนาคาร

การวาดอ๊อบเจ็กต์ในลักษณะนี้จะทำให้เข้าใจได้กระจ่างขึ้นว่า ตัวแปรประจำอ๊อบเจ็กต์ของแต่ละอ๊อบเจ็กต์ จะถูกเก็บในที่ที่ต่างกัน เมทอดประจำอ๊อบเจ็กต์ของแต่ละอ๊อบเจ็กต์จะประมวลผลตัวแปรประจำอ๊อบเจ็กต์ของตนเองเท่านั้น ไม่ยุ่งเกี่ยวกับอ๊อบเจ็กต์อื่น

## 1.9 การจัดแบ่งโปรแกรมเป็นคลาส

ดังที่กล่าวไว้ตอนต้นของบทนี้ คุณภาพของซอฟต์แวร์มีความสำคัญ และไม่ว่าจะเป็นเรื่องใหญ่ เช่น การแบ่งโปรแกรมเป็นคลาสให้เหมาะสม การกระจายข้อมูลและเมทอดให้อยู่ในคลาสที่เหมาะสม หรือเรื่องเล็กน้อย เช่น การตั้งชื่อคลาส ตัวแปร และเมทอด การตัดสินใจเหล่านี้จะส่งผลต่อคุณภาพของซอฟต์แวร์ได้ หากแบ่งคลาสไม่เหมาะสม การปรับปรุงแก้ไขคลาสหนึ่งอาจส่งผลกระทบต่อคลาสอื่น ทำให้ต้องแก้ไขตามไปด้วย การตั้งชื่อที่ไม่สื่อถึงการใช้งานจะทำให้โปรแกรมเมอร์ต้องใช้เวลานานในการทำความเข้าใจการทำงานของโปรแกรม ความไม่เหมาะสมทั้งสองนี้ ทำให้โปรแกรมเข้าใจได้ยาก ปรับปรุงแก้ไขโปรแกรมได้ยาก และมีโอกาสทำงานผิดพลาดได้สูง หัวข้อนี้จะอธิบายการแบ่งโปรแกรมเป็นคลาสเบื้องต้น และหัวข้อถัดไปจะอธิบายหลักการตั้งชื่อคลาส ตัวแปร และเมทอด เพื่อให้โปรแกรมมีคุณลักษณะที่ดี

## หลักการจัดแบ่งโปรแกรมเป็นคลาส

เนื่องจากคลาสถือเป็นประเภทของข้อมูล เป็นแม่พิมพ์เพื่อสร้างอ๊อบเจ็กต์หรือวัตถุ จึงต้องสามารถมีข้อมูลที่เป็นประเภทของคลาสนี้ได้หลายข้อมูล หรือสร้างอ๊อบเจ็กต์หรือวัตถุที่เป็นประเภทของคลาสนี้ได้ออกมาได้หลายอ๊อบเจ็กต์ นอกจากนั้น คลาสเป็นที่รวบรวมและห่อหุ้มข้อมูลกับเมทอดที่สัมพันธ์กันไว้ภายใน คลาสจึงควรจะเป็นตัวแทนของกลุ่มข้อมูลและเมทอดเหล่านั้น และเราควรตั้งชื่อคลาสให้ตรงกับบริบทการใช้งานกลุ่มข้อมูลและเมทอดนั้น ๆ

เมื่อแบ่งโปรแกรมเป็นคลาสได้แล้ว เราต้องวิเคราะห์ด้วยว่า ในคลาสต้องประกอบด้วยตัวแปรประจำอ๊อบเจ็กต์ใดบ้าง หัวข้อ 1.2 อธิบายแล้วว่าตัวแปรประจำอ๊อบเจ็กต์ควรเป็นคุณลักษณะของอ๊อบเจ็กต์ ดังนั้น เราต้องหาคุณลักษณะเหล่านี้ให้ได้ โดยต้องเป็นคุณลักษณะที่จำเป็นในการประมวลผลโปรแกรมด้วย สำหรับเมทอดประจำอ๊อบเจ็กต์ ต้องเลือกเมทอดที่เกี่ยวข้องกับคลาส ตรงตามพฤติกรรม การทำงาน หรือหน้าที่ที่อ๊อบเจ็กต์ของคลาสนั้นสามารถทำได้ หรือเป็นการทำงานที่ให้ผู้อื่นใช้งานคลาสนี้ได้ โดยแต่ละเมทอดควรทำหน้าที่อย่างใดอย่างหนึ่งที่ชัดเจน

## ตัวอย่างการจัดแบ่งโปรแกรมเป็นคลาส

โดยสรุป เมื่อแบ่งโปรแกรมเป็นคลาส เราต้องวิเคราะห์ 4 อย่างต่อไปนี้ให้ได้

1. คลาส
    a) ตัวแปรประจำอ๊อบเจ็กต์
    b) เมทอดประจำอ๊อบเจ็กต์
2. อ๊อบเจ็กต์

ดังตัวอย่างต่อไปนี้

- การคำนวณพื้นที่และความยาวรอบรูปของสามเหลี่ยม สามารถทำให้เป็นคลาสได้
    1. คลาส: ตั้งชื่อคลาสว่าสามเหลี่ยม (`Triangle`)
        a) ตัวแปรประจำอ๊อบเจ็กต์: สามเหลี่ยมมีคุณลักษณะที่เกี่ยวข้องหลายอย่าง เช่น ความยาวของแต่ละด้าน ความยาวฐาน ความสูง ขนาดของมุมแต่ละมุม เป็นต้น สำหรับในบริบทนี้ เราต้องการคำนวณพื้นที่และความยาวรอบรูปของสามเหลี่ยม จึงควรมีตัวแปรความยาวของแต่ละด้านเป็นตัวแปรประจำอ๊อบเจ็กต์ เพื่อให้คำนวณความยาวรอบรูปและคำนวณพื้นที่โดยใช้สูตรของเฮรอนได้ สำหรับความสูงและขนาดของมุมแต่ละมุมจะไม่จำเป็นในกรณีนี้ อย่างไรก็ตาม หากต้องการ

ตรวจสอบว่าเป็นสามเหลี่ยมประเภทใด เช่น สามเหลี่ยมมุมฉาก สามเหลี่ยมมุมป้าน จะต้องมีขนาดของมุมเป็นตัวแปรประจำอ๊อบเจ็กต์ด้วย

b) เมทอดประจำอ๊อบเจ็กต์: สามารถคำนวณพื้นที่และความยาวรอบรูปของสามเหลี่ยมได้ จึงควรมีเมทอดประจำอ๊อบเจ็กต์ `getArea()` และ `getPerimeter()`

2. อ๊อบเจ็กต์: คลาสสามเหลี่ยมนี้สามารถสร้างอ๊อบเจ็กต์หรือวัตถุได้หลากหลาย เช่น สามเหลี่ยมที่มีความยาวแต่ละด้านเป็น 2, 2, 2 หรือ สามเหลี่ยมที่มีความยาวแต่ละด้านเป็น 3, 4, 5 เป็นต้น

- การเก็บและการประมวลผลข้อมูลเกี่ยวกับหนังสือ เพื่อใช้ในโปรแกรมการยืมและคืนหนังสือในห้องสมุด
    1. คลาส: ตั้งชื่อคลาสว่าหนังสือ (`Book`)
        a) ตัวแปรประจำอ๊อบเจ็กต์: หนังสือมีคุณลักษณะที่เกี่ยวข้อง เช่น ชื่อหนังสือ ชื่อผู้แต่ง สำนักพิมพ์ ราคา โดยในบริบทของโปรแกรมการยืมและคืนหนังสือ ควรมีตัวแปร `title`, `author`, `publisher` เป็นตัวแปรประจำอ๊อบเจ็กต์ แต่ไม่จำเป็นต้องมีราคา เนื่องจากไม่มีการซื้อขาย นอกจากนั้น ควรมีตัวแปร `status` เป็นตัวแปรประจำ อ๊อบเจ็กต์ เพื่อแสดงถึงสถานะการยืมคืนหนังสือในห้องสมุดด้วยถึงแม้ว่าจะไม่ใช่คุณลักษณะโดยตรงของหนังสือทั่วไป
        b) เมทอดประจำอ๊อบเจ็กต์: สามารถยืมและคืนหนังสือในห้องสมุดได้ จึงควรมีเมทอดประจำอ๊อบเจ็กต์ `borrow()` และ `bringBack()`
    2. อ๊อบเจ็กต์: คลาสหนังสือนี้สามารถสร้างอ๊อบเจ็กต์หรือวัตถุได้หลากหลาย เช่น หนังสือชื่อ C++ แต่งโดยคุณเบียเนอ สเดราสดร็อบ หรือหนังสือชื่อ Big Java แต่งโดย Cay S. Horstmann เป็นต้น
- การซื้อขายสินค้า เพื่อใช้ในโปรแกรม e-commerce
    1. คลาส: ตั้งชื่อคลาสว่าสินค้า (`Product`)
        a) ตัวแปรประจำอ๊อบเจ็กต์: สินค้ามีคุณลักษณะที่เกี่ยวข้อง เช่น ชื่อสินค้า ราคา จำนวนในสต๊อค จึงควรมีตัวแปร `name`, `price` และ `quantity` เป็นตัวแปรประจำอ๊อบเจ็กต์
        b) เมทอดประจำอ๊อบเจ็กต์: สามารถซื้อและขายสินค้าได้ จึงควรมีเมทอดประจำอ๊อบเจ็กต์ `buy()` และ `sell()`
    2. อ๊อบเจ็กต์: คลาสสินค้านี้สามารถสร้างอ๊อบเจ็กต์หรือวัตถุได้หลากหลาย เช่น สินค้าชื่อน้ำส้ม ราคา 15 บาท มีจำนวน 50 กล่องในสต๊อค หรือสินค้าชื่อ สมุด ราคา 32 บาท มีจำนวน 80 เล่มในสต๊อค เป็นต้น

## 1.10 การตั้งชื่อคลาส ตัวแปร และเมทอด

การตั้งชื่อคลาส ตัวแปร และเมทอดมีความสำคัญ หากตั้งชื่อที่สื่อถึงการทำงานของคลาส ตัวแปร หรือเมทอดได้ดี จะสามารถช่วยให้โปรแกรมเข้าใจง่ายโดยไม่จำเป็นต้องเขียนคอมเม้นต์เพิ่มเติม การตั้งชื่อต่าง ๆ เหล่านี้จึงมีประเพณีนิยมการตั้งชื่อ (naming convention) ที่ถือว่าเป็นสากล ไม่ว่าจะเป็นโปรแกรมเมอร์จากชาติใด ก็จะใช้ประเพณีนิยมการตั้งชื่อเดียวกัน ดังนั้น เมื่อเขียนโปรแกรม เราจึงควรจะตั้งชื่อตามประเพณีนิยม ภาษาโปรแกรมแต่ละภาษาจะมีประเพณีนิยมการตั้งชื่อที่แตกต่างกัน แต่เป็นสากลสำหรับภาษาโปรแกรมนั้น ๆ หัวข้อนี้อธิบายประเพณีนิยมการตั้งชื่อในภาษาจาวา

### คลาส

จากตัวอย่างในหัวข้อ 1.9 และตัวอย่างคลาสสี่เหลี่ยม คลาสบัญชีธนาคาร และคลาสนักเรียน จะเห็นว่าชื่อคลาสล้วนแต่เป็นคำนาม เนื่องจากคลาสเป็นต้นแบบของวัตถุซึ่งเป็นคำนาม ดังนั้น การตั้งชื่อคลาสควรเป็น**คำนาม**เอกพจน์ เนื่องจากคลาสเป็นประเภทข้อมูลที่ใช้เป็นต้นแบบของวัตถุหนึ่ง ๆ ไม่ใช่หลายวัตถุ

ในภาษาจาวา ชื่อคลาสจะขึ้นต้นด้วยตัวอักษรภาษาอังกฤษตัวใหญ่ ตัวอักษรอื่นในคำนามจะเป็นตัวอักษรเล็ก หากชื่อคลาสประกอบด้วยคำมากกว่า 1 คำ แต่ละคำจะขึ้นต้นด้วยตัวอักษรใหญ่ เช่น คลาสหนังสือ (`Book`) คลาสบัญชีธนาคาร (`BankAccount`) คลาสตารางหมากรุก (`ChessBoard`) เราเรียกรูปแบบการตั้งชื่อในลักษณะนี้ว่า หลังอูฐ (camel case) เนื่องจากมีตัวอักษรตัวเล็กตัวใหญ่สลับกันไปคล้ายโหนกบนหลังอูฐ

### ตัวแปร

สำหรับการตั้งชื่อตัวแปรที่ใช้ในการเก็บข้อมูลนั้น ทั้งชื่ออ๊อบเจ็กต์ ชื่อตัวแปรประจำอ๊อบเจ็กต์ ชื่อตัวแปรโลคอล (local variable) และชื่อพารามิเตอร์ ควรจะตั้งให้ตรงกับบริบทการใช้งานของข้อมูล นิยมตั้งเป็น**คำนาม**ซึ่งอาจจะเป็นเอกพจน์หรือพหุพจน์ ขึ้นอยู่กับว่า ตัวแปรนั้น ๆ เก็บข้อมูลเป็นจำนวนเท่าใด เช่น ตัวแปรที่เป็นอาเรย์จะมีชื่อเป็นคำนามพหุพจน์ เนื่องจากอาเรย์จะเก็บข้อมูลจำนวนหลายข้อมูลในตัวแปรเดียวกัน ส่วนตัวแปรอื่นที่ไม่ใช่อาเรย์จะมีชื่อเป็นคำนามเอกพจน์

การตั้งชื่อตัวแปรในภาษาจาวา จะใช้รูปแบบหลังอูฐคล้ายกับการตั้งชื่อคลาส แต่จะขึ้นต้นคำแรกด้วยตัวอักษรเล็ก หากชื่อตัวแปรประกอบด้วยคำมากกว่า 1 คำ คำต่อ ๆ มาจะขึ้นต้นด้วยตัวอักษรใหญ่ เช่น `midtermScore`, `firstName`

### เมทอดประจำอ๊อบเจ็กต์

การตั้งชื่อเมทอดประจำอ๊อบเจ็กต์ในภาษาจาวา ควรตั้งให้ตรงกับบริบทการใช้งาน โดยนิยมตั้งเป็น**คำกริยา**ที่เป็นพฤติกรรมของคลาสนั้น ๆ และใช้รูปแบบหลังอูฐเช่นเดียวกับการตั้งชื่อตัวแปร คือ คำแรกขึ้นต้นด้วยตัวอักษรภาษาอังกฤษตัวเล็ก คำต่อ ๆ มา จะขึ้นต้นด้วยตัวอักษรใหญ่ เช่น `getTotalScore()`

เมื่อเรียกเมทอดประจำอ๊อบเจ็กต์จากชื่ออ๊อบเจ็กต์ จะทำให้รหัสคำสั่งมีลักษณะเหมือนประโยคบอกเล่า กล่าวคือ ชื่ออ๊อบเจ็กต์ซึ่งเป็นคำนามจะเป็นประธานของประโยค ชื่อเมทอดประจำอ๊อบเจ็กต์จะเป็นกริยาของประโยค และพารามิเตอร์ซึ่งเป็นคำนามจะเป็นกรรมของประโยค ส่งผลให้รหัสคำสั่งมีความคล้ายคลึงกับภาษาธรรมชาติ ทำให้โปรแกรมอ่านเข้าใจได้ง่ายขึ้น

## 1.11 แบบฝึกหัด

1. จงเขียนโปรแกรมเพื่อยืมคืนหนังสือในห้องสมุด โดยหนังสือมีข้อมูล ชื่อหนังสือ ชื่อผู้แต่ง สำนักพิมพ์ และจำนวนหนังสือที่ห้องสมุดมีให้ยืม เช่น หนังสือชื่อ C Programming ผู้แต่งคือ Dennis Richie สำนักพิมพ์ McGraw-Hill และห้องสมุดมีหนังสือนี้อยู่ 3 เล่มที่จะให้ยืมได้

   นักเรียนสามารถยืมหนังสือได้ โดยยืมได้ครั้งละหนึ่งเล่ม เมื่อนักเรียนยืมหนังสือแต่ละครั้ง จำนวนหนังสือที่ห้องสมุดมีให้ยืมจะลดลงเรื่อย ๆ หากนักเรียนหลายคนยืมหนังสือเล่มนี้ออกไปจนหมด นักเรียนคนถัดไปจะยืมหนังสือเล่มนั้นไม่ได้ เมื่อนักเรียนเหล่านั้นคืนหนังสือ จำนวนหนังสือที่ห้องสมุดมีให้ยืมจะเพิ่มขึ้น

   โดยให้โปรแกรมตามข้อกำหนดต่อไปนี้

   - โปรแกรมต้องมีตัวแปรและเมทอดประจำอ๊อบเจ็กต์ที่เหมาะสมกับการยืมคืนหนังสือ
   - โปรแกรมต้องสร้างอ๊อบเจ็กต์ของหนังสือขึ้นมาอย่างน้อย 2 เล่ม และให้เรียกเมทอดเพื่อยืมและคืนหนังสือ

# บทที่ 2

# เมทอดและคอนสตรักเตอร์

การนิยามคลาสให้รวบรวมและห่อหุ้มข้อมูลที่เกี่ยวข้องกันไว้ด้วยกันเป็นตัวแปรประจำอ๊อบเจ็กต์ และให้คลาสมีเมทอดประจำอ๊อบเจ็กต์เพื่อประมวลผลตัวแปรประจำอ๊อบเจ็กต์ รวมถึง การสร้างอ๊อบเจ็กต์ของคลาส เป็นแก่นสำคัญของการโปรแกรมเชิงวัตถุ นอกจากแก่นแล้ว กลไกและหลักการอื่นที่จะช่วยให้โปรแกรมมีคุณลักษณะที่ดีขึ้น บทนี้จะอธิบายกลไกและหลักการเหล่านี้

## 2.1 การโอเวอร์โหลดเมทอดและคอนสตรักเตอร์

ภาษาจาวาอนุญาตให้คลาสหนึ่ง ๆ สามารถมีเมทอดชื่อเดียวกันได้หลายเมทอด โดยเมทอดเหล่านี้ต้องรับพารามิเตอร์เข้ามาต่างกัน เราเรียกการอนุญาตในลักษณะนี้ว่า การโอเวอร์โหลดเมทอด (method overload) นอกจากนั้น ยังอนุญาตให้คลาสหนึ่ง ๆ สามารถมีคอนสตรักเตอร์ได้หลายตัว เรียกว่า การโอเวอร์โหลดคอนสตรักเตอร์ (constructor overload) โดยคอนสตรักเตอร์แต่ละตัวจะต้องรับพารามิเตอร์เข้ามาต่างกัน

พารามิเตอร์จะต่างกันได้ 2 กรณี ในดังนี้

- มีจำนวนพารามิเตอร์ต่างกัน
- ถ้าจำนวนพารามิเตอร์เท่ากัน อย่างน้อยหนึ่งพารามิเตอร์ต้องมีประเภทข้อมูลต่างกัน

หัวข้อนี้จะอธิบายการโอเวอร์โหลดเมทอดและการโอเวอร์โหลดคอนสตรักเตอร์

## การโอเวอร์โหลดเมทอด

เมทอดของอ๊อบเจ็กต์แสดงถึงพฤติกรรมการทำงานของอ๊อบเจ็กต์ โดยเมทอดสามารถรับข้อมูลที่เกี่ยวข้องกับการกระทำนั้น ๆ เข้ามาทางพารามิเตอร์ของเมทอดได้ ข้อมูลเหล่านี้อาจมีได้หลากหลายรูปแบบ เช่น คลาสสี่เหลี่ยมอาจมีเมทอด `changeSize` เพื่อปรับขนาดความกว้างและความยาวของอ๊อบเจ็กต์สี่เหลี่ยม โดยอาจปรับทั้งความกว้างและความยาวด้วยค่าเดียวกันหรือด้วยค่าที่ต่างกันได้ แทนที่จะนิยามเมทอด `changeSize` เมทอดเดียวด้วยพารามิเตอร์สองค่า โดยพารามิเตอร์แรกปรับความกว้างและพารามิเตอร์ที่สองปรับความยาว เราสามารถนิยามเมทอด `changeSize` ขึ้นมาอีกเมทอดด้วยชื่อเดียวกัน แต่รับเพียงแค่พารามิเตอร์เดียวเพื่อใช้ปรับทั้งความกว้างและความยาวได้

ตัวอย่างคลาสสี่เหลี่ยมด้านล่างนิยามเมทอดประจำอ๊อบเจ็กต์ `changeSize` มาสองเมทอด เพื่อปรับขนาดความกว้างและความยาวของอ๊อบเจ็กต์สี่เหลี่ยม โดยเมทอด `ChangeSize` ทั้งสองนี้มีจำนวนพารามิเตอร์ต่างกัน เมทอดแรกรับพารามิเตอร์ค่าเดียว คือ ค่า `d` เพื่อปรับทั้งความกว้างและความยาว ส่วนเมทอดที่ 2 รับสองพารามิเตอร์ `dw` และ `dl` เพื่อปรับค่าความกว้างและความยาวตามลำดับ

```
1  class Rectangle {
2
3      private double width;
4      private double length;
5
6      public Rectangle(double width, double length) {
7          this.width = width;
8          thislength = length;
9      }
10
11     public double area() {
12         return width * length;
13     }
14
15     public double perimeter() {
16         return 2 * (width + length);
17     }
18
19     // ******* method overload *******
20     public void changeSize(int d) {
21         width += d;
22         length += d;
23     }
24
25     public void changeSize(int dw, int dl) {
```

```
26          width += dw;
27          length += dl;
28      }
29  }
```

ดังนั้น เราสามารถเรียกใช้เมทอด `changeSize` ได้สองรูปแบบดังรหัสคำสั่งต่อไปนี้ บรรทัดแรกสร้างอ๊อบเจ็กต์สี่เหลี่ยมมา 2 อ๊อบเจ็กต์ `square` และ `rect` บรรทัดที่ 2 เรียกเมทอด `changeSize` แบบพารามิเตอร์เดียวเพื่อเปลี่ยนความกว้างและความยาวด้วยค่าเดียวกัน บรรทัดที่ 3 เรียกเมทอด `changeSize` แบบสองพารามิเตอร์

```
1  Rectangle square = new Rectangle(4,4);
2  Rectangle rect = new Rectangle(6,8);
3  square.changeSize(1);
4  rect.changeSize(2,3);
```

การโอเวอร์โหลดเมทอดจะทำให้โปรแกรมมีความยืดหยุ่น สามารถใช้เมทอดชื่อเดียวกันเพื่อทำงานได้หลากหลายรูปแบบ

ตัวอย่างการโอเวอร์โหลดเมทอดในคลาสสี่เหลี่ยมนี้เป็นการโอเวอร์โหลดแบบมีจำนวนพารามิเตอร์ที่ต่างกัน การโอเวอร์โหลดสามารถทำได้เมื่อจำนวนพารามิเตอร์เท่ากัน แต่มีพารามิเตอร์อย่างน้อยหนึ่งตัวที่มีประเภทที่ต่างกัน โดยตัวอย่างจะอยู่ในหัวข้อการโอเวอร์โหลดคอนสตรักเตอร์

## การโอเวอร์โหลดคอนสตรักเตอร์

คลาสหนึ่ง ๆ สามารถมีคอนสตรักเตอร์ได้หลายตัว เรียกว่า การโอเวอร์โหลดคอนสตรักเตอร์ (constructor overload) คอนสตรักเตอร์มีประโยชน์ในการช่วยบังคับให้มีการกำหนดค่าตัวแปรประจำอ๊อบเจ็กต์ที่จำเป็น อย่างไรก็ตาม อาจมีตัวแปรประจำอ๊อบเจ็กต์บางตัวที่สามารถใช้ค่าปริยาย (default value) ได้ ทำให้ผู้ใช้เลือกที่จะกำหนดหรือไม่กำหนดค่าตัวแปรประจำอ๊อบเจ็กต์เหล่านี้ก็ได้ ภาษาจาวาจึงอนุญาตให้คลาสหนึ่ง ๆ สามารถมีคอนสตรักเตอร์ได้หลายตัว เมื่อโปรแกรมเมอร์เลือกที่จะไม่กำหนดค่าให้กับตัวแปรประจำอ๊อบเจ็กต์ จะสามารถเลือกใช้คอนสตรักเตอร์ที่กำหนดค่าปริยายให้กับตัวแปรประจำอ๊อบเจ็กต์นั้น ๆ ได้ การโอเวอร์โหลดจึงช่วยให้โปรแกรมมีความยืดหยุ่นมากขึ้น

คลาสบัญชีธนาคารในรหัสคำสั่งด้านล่าง มีคอนสตรักเตอร์ 2 ตัว ซึ่งมีจำนวนพารามิเตอร์ต่างกัน โดยคอนสตรักเตอร์ตัวแรกรับทั้งชื่อบัญชีธนาคาร `n` และยอดคงเหลือ `b` เข้ามาผ่านพารามิเตอร์และกำหนดค่าเหล่านี้ให้กับตัวแปรประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ตัวที่ 2 รับเพียงแค่ชื่อบัญชี `n` เข้ามาทางพารามิเตอร์

เท่านั้น และกำหนดค่าปริยาย 0 ให้กับตัวแปรประจำอ๊อบเจ็กต์ยอดคงเหลือ คอนสตรักเตอร์ทั้งสองตัวนี้ทำให้โปรแกรมมีความยืดหยุ่น จะกำหนดค่ายอดคงเหลือในบัญชีเริ่มต้นหรือไม่ก็ได้

```
1  class BankAccount {
2
3      // ---- instance variables ----
4      private String name;
5      private double balance;
6
7      // ---- constructors ----
8      public BankAccount(String name, double balance) {
9          this.name = name;
10         this.balance = balance;
11     }
12
13     public BankAccount(String name) {
14         this.name = n;
15         this.balance = 0;
16     }
17
18     // ---- instance methods ----
19     public void deposit(double amount) {
20         if (amount > 0)
21             balance += amount;
22     }
23     public void withdraw(double amount) {
24         if (amount > 0 && amount < balance)
25             balance -= amount;
26     }
27     // ---- getters and setters ---
28     public String getName() {
29         return name;
30     }
31     public double getBalance() {
32         return balance;
33     }
34     public void setName(String name) {
35         this.name = name;
36     }
37 }
```

รหัสคำสั่งข้างต้นจะช่วยให้สามารถสร้างอ๊อบเจ็กต์ได้สองแบบดังรหัสคำสั่งต่อไปนี้

```
BankAccount annAct = new BankAccount("Ann", 1000);
BankAccount benAct = new BankAccount("Ben");
```

รหัสคำสั่งนี้สร้างอ๊อบเจ็กต์บัญชีธนาคารขึ้นมาสองอ๊อบเจ็กต์ คือ annAct และ benAct โดยการสร้าง อ๊อบเจ็กต์ annAct จะใช้คอนสตรักเตอร์ตัวแรกที่รับพารามิเตอร์มาสองตัวในการกำหนดค่าดาต้าเมมเบอร์ทั้งสอง และการสร้างอ๊อบเจ็กต์ benAct จะใช้คอนสตรักเตอร์ตัวที่สอง ทำให้ค่ายอดคงเหลือของอ๊อบเจ็กต์ benAct จะเป็นค่า 0 ซึ่งเป็นค่าปริยาย

ตัวอย่างการโอเวอร์โหลดคอนสตรักเตอร์ในคลาสบัญชีธนาคารมีจำนวนพารามิเตอร์ที่ต่างกัน ตัวอย่างคลาสหนังสือต่อไปนี้ แสดงการโอเวอร์โหลดคอนสตรักเตอร์ที่มีจำนวนพารามิเตอร์เท่ากัน แต่มีประเภทของพารามิเตอร์ที่ต่างกันหนึ่งตัว คอนสตรักเตอร์ตัวที่ 1 รับพารามิเตอร์ประเภทสตริงทั้งสองพารามิเตอร์ เพื่อนำไปกำหนดค่าให้ตัวแปรประจำอ๊อบเจ็กต์ชื่อหนังสือ name และสำนักพิมพ์ publisher โดยให้จำนวนหนังสือ quantity มีค่าปริยายเป็น 1 ขณะที่คอนสตรักเตอร์ตัวที่ 2 รับพารามิเตอร์ประเภทสตริงและจำนวนเต็ม เพื่อกำหนดค่าให้ตัวแปรประจำอ๊อบเจ็กต์ชื่อหนังสือ name และจำนวน quantity โดยให้สำนักพิมพ์ publish มีค่าปริยายเป็น "KU"

```
1  class Book {
2
3      // ---- instance variables ----
4      private String name;
5      private String publisher;
6      private double quantity;
7
8      // ---- constructors ----
9      public Book(String name, String publisher) {
10         this.name = name;
11         this.publisher = publisher;
12         this.quantity = 1;
13     }
14     public Book(String name, int quantity) {
15         this.name = name;
16         this.publisher = "KU";
17         this.quantity = quantity;
18     }
19
20     // ---- instance methods ----
21     public void purchase(int amount) {
22         this.quantity += amount;
23     }
24 }
```

อนึ่ง หากคอนสตรักเตอร์สองตัวมีจำนวนพารามิเตอร์เท่ากัน และทุกพารามิเตอร์เป็นประเภทเดียวกัน แต่มีชื่อตัวแปรพารามิเตอร์ที่ต่างกัน จะไม่ถือว่าเป็นการโอเวอร์โหลดคอนสตรักเตอร์ หากทำเช่นนั้น คอมไพเลอร์จะแจ้งเตือนว่า มีการนิยามคอนสตรักเตอร์ซ้ำกัน รหัสคำสั่งต่อไปนี้แสดงการโอเวอร์โหลดคอนสตรักเตอร์ที่ไม่ถูกต้อง

```
class Student {

    // ---- instance variables ----
    private String name;
    private int midtermScore, finalScore;

    // ---- constructors : INCORRECT overload ----
    public Student(String name, int midtermScore) {
        this.name = name;
        this.midtermScore = midtermScore;
        this.finalScore = 0;
    }

    public Student(String name, int ffinalScore) {
        this.name = name;
        this.midtermScore = 0;
        this.finalScore = finalScore;
    }

    // ---- instance methods ----
    public int totalScore() {
        return midtermScore + finalScore;
    }
}
```

ในรหัสคำสั่งข้างต้น คอนสตรักเตอร์รับค่าสตริงและจำนวนเต็มมาทั้งสองตัว ถึงแม้จะมีชื่อพารามิเตอร์ต่างกัน เมื่อรันคอมไพล์เลอร์ จะมีการแจ้งเตือนข้อผิดพลาดว่าคอนสตรักเตอร์ซ้ำกันดังนี้

```
'Student(String, int)' is already defined in 'Student'
```

## 2.2 คอนสตรักเตอร์แบบปริยาย

คอนสตรักเตอร์ที่ไม่รับพารามิเตอร์มีชื่อเรียกพิเศษว่า คอนสตรักเตอร์แบบปริยาย (default constructor) ในภาษาจาวา หากเราไม่ได้มีการนิยามคอนสตรักเตอร์ใด ๆ จาวาจะสร้างคอนสตรักเตอร์แบบปริยายให้โดยอัตโนมัติ โดยคอนสตรักเตอร์ที่สร้างขึ้นมาโดยอัตโนมัตินี้ จะทำหน้าที่สร้างอ๊อบเจ็กต์ และกำหนดเริ่มต้นเป็นค่าโดยปริยายให้กับตัวแปรประจำอ๊อบเจ็กต์ จึงทำให้ในบทที่ 1 - 2 เราสามารถสร้างอ๊อบเจ็กต์ได้โดยไม่ต้องมีพารามิเตอร์ได้ ทั้งที่ยังไม่ได้นิยามคอนสตรักเตอร์

แต่ถ้าเรานิยามคอนสตรักเตอร์เองแล้ว จาวาจะไม่สร้างคอนสตรักเตอร์แบบปริยายให้โดยอัตโนมัติ จะต้องสร้างอ๊อบเจ็กต์ตามคอนสตรักเตอร์ที่เรานิยามไว้เท่านั้น ดังนั้น หากเราประกาศคอนสตรักเตอร์แบบมีพารามิเตอร์ เราจะสร้างอ๊อบเจ็กต์แบบไม่ใส่พารามิเตอร์ไม่ได้

อนึ่ง เราสามารถนิยามคอนสตรักเตอร์แบบปริยายเองได้ โดยทั่วไป เราจะนิยามคอนสตรักเตอร์แบบปริยายเองเมื่อตัวแปรประจำอ๊อบเจ็กต์บางตัวมีค่าโดยปริยาย และจะโอเวอร์โหลดคอนสตรักเตอร์แบบปริยายร่วมกับคอนสตรักเตอร์อื่น

ตัวอย่างคลาส Rational ในรหัสคำสั่งต่อไปนี้แสดงการนิยามคอนสตรักเตอร์แบบปริยาย โดยคลาส Rational เป็นประเภทข้อมูลที่เป็นจำนวนตรรกยะหรือจำนวนเศษส่วน มีตัวแปรประจำอ๊อบเจ็กต์สองค่า คือ เศษ (numerator) และส่วน (denominator) คลาสนี้มีคอนสตรักเตอร์สองตัว ตัวแรกเป็นคอนสตรักเตอร์แบบปริยายที่อนุญาตให้สร้างอ๊อบเจ็กต์โดยไม่ต้องกำหนดทั้งเศษและส่วน โดยคอนสตรักเตอร์จะกำหนดค่าโดยปริยายให้เศษเป็น 0 และส่วนเป็น 1 หมายถึง จำนวนเต็มศูนย์ ส่วนคอนสตรักเตอร์ตัวที่สองจะรับค่าทั้งเศษและส่วนเข้ามากำหนดค่าให้ตัวแปรประจำอ๊อบเจ็กต์ทั้งสอง

```
1  class Rational {
2      private int numerator;
3      private int denominator;
4
5      public Rational() {
6          numerator = 0;
7          denominator = 1;
8      }
9
10     public Rational(int numerator, int denominator) {
11         this.numerator = numerator;
12         this.denominator = denominator;
13     }
14
15     // ..... instance methods .....
16 }
```

รหัสคำสั่งข้างต้นจะช่วยให้สามารถสร้างอ๊อบเจ็กต์ได้สองแบบดังรหัสคำสั่งต่อไปนี้

```
Rational zero = new Rational();
Rational half = new Rational(1,2);
```

## 2.3 ประเภทของเมทอดประจำอ๊อบเจ็กต์

เราสามารถแบ่งเมทอดประจำอ๊อบเจ็กต์เป็น 2 ประเภท ตามการประมวลผลตัวแปรประจำอ๊อบเจ็กต์ ดังนี้

- แอคเซสเซอร์ (accessor) เป็นเมทอดที่อ่าน คำนวณ และคืนค่าตัวแปรประจำอ๊อบเจ็กต์ โดยไม่มีการเปลี่ยนแปลงค่าตัวแปรประจำอ๊อบเจ็กต์ใด ๆ
- มิวเทเทอร์ (mutator) เป็นเมทอดที่กำหนดหรือเปลี่ยนแปลงค่าตัวแปรประจำอ๊อบเจ็กต์

เมทอดประจำอ๊อบเจ็กต์ `area()` ในคลาสสี่เหลี่ยมถือว่าเป็นเมทอดประเภทแอคเซสเซอร์ เนื่องจากนำค่าความกว้างและความยาวของสี่เหลี่ยมมาคำนวณหาพื้นที่เท่านั้น ไม่มีการเปลี่ยนแปลงค่าทั้งสอง เมทอดประจำอ๊อบเจ็กต์ `totalScore()` ในคลาสนักเรียนก็ถือว่าเป็นเมทอดประเภทแอคเซสเซอร์เช่นกัน มีการนำตัวแปรประจำอ๊อบเจ็กต์คะแนนมาคำนวณผลรวม โดยไม่มีการเปลี่ยนแปลงค่าคะแนน เมทอดประเภทแอคเซสเซอร์ถือเป็นเมทอดที่ปลอดภัย เมื่อคลาสอื่นนำอ๊อบเจ็กต์ไปใช้ จะไม่มีการเปลี่ยนแปลงค่าตัวแปรประจำอ๊อบเจ็กต์

```
// ---- accessor ----
public double area() {
    return width * length;
}
```

```
// ---- accessor ----
public int totalScore() {
    return midtermScore + finalScore;
}
```

ส่วนเมทอดประจำอ๊อบเจ็กต์ `deposit()` และ `withdraw()` ในคลาสบัญชีธนาคารถือว่าเป็นเมทอดประเภทมิวเทเทอร์ เนื่องจากมีการเปลี่ยนแปลงค่าตัวแปรประจำอ๊อบเจ็กต์ยอดคงเหลือในบัญชีธนาคาร เมทอดประเภทมิวเทเทอร์อาจจะอ่าน คำนวณ และคืนค่าตัวแปรประจำอ๊อบเจ็กต์หรือไม่ก็ได้ แต่จะเป็นมิวเทเทอร์เมื่อมีการเปลี่ยนแปลงค่าตัวแปรประจำอ๊อบเจ็กต์ เราถือว่าเมทอดประเภทมิวเทเทอร์เป็นเมทอดที่ไม่ค่อยปลอดภัย เนื่องจากคลาสอื่นสามารถปรับเปลี่ยนค่าตัวแปรภายในอ๊อบเจ็กต์ผ่านเมทอดนี้ได้

```
// ---- mutator ----
public void deposit(double amount) {
    balance = balance + amount;
}
// ---- mutator ----
public void withdraw(double amount) {
    balance = balance - amount;
}
```

ภาษาจาวาไม่มีคีย์เวิร์ดที่ใช้ในการระบุว่าเมทอดนี้เป็นประเภทแอคเซสเซอร์หรือมิวเทเทอร์ได้ ภาษาโปรแกรมอื่น เช่น ภาษา C++ มีคีย์เวิร์ดที่ช่วยระบุว่าเมทอดเป็นประเภทแอคเซสเซอร์ และตัวคอมไพล์จะช่วยตรวจสอบด้วยว่า เมทอดที่มีคีย์เวิร์ดนี้จะไม่สามารถกำหนดหรือเปลี่ยนแปลงค่าตัวแปรประจำอ๊อบเจ็กต์ได้

ดังนั้น โปรแกรมเมอร์ภาษาจาวาจะต้องระมัดระวังการเปลี่ยนแปลงค่าตัวแปรประจำอ๊อบเจ็กต์เองโดยอาจการใช้การควบคุมการเข้าถึงเมทอดมาช่วย ดังจะอธิบายในบทถัดไป

# บทที่ 3

# ความสัมพันธ์ระหว่างคลาส

เมื่อโปรแกรมมีขนาดใหญ่ขึ้น การเขียนรหัสคำสั่งทั้งหมดไว้ในคลาสเดียวจะทำให้โปรแกรมอ่านเข้าใจยาก มีโอกาสทำงานผิดพลาดได้สูง จึงควรแบ่งโปรแกรมออกเป็นหลายคลาส แล้วจึงนำคลาสเหล่านั้นมาเชื่อมต่อกันและทำงานร่วมกันให้บรรลุวัตถุประสงค์ที่ต้องการ การเชื่อมต่อคลาสจะเป็นการสร้างความสัมพันธ์ระหว่างคลาส โดยความสัมพันธ์ระหว่างคลาสในการโปรแกรมเชิงวัตถุมีหลายประเภท และหนังสือเกี่ยวกับการโปรแกรมและพัฒนาซอฟต์แวร์อาจแบ่งประเภทไม่ตรงกันนัก (แต่มีความคล้ายคลึงกัน) หนังสือเล่มนี้แบ่งประเภทความสัมพันธ์ออกเป็น 3 ประเภทหลัก คือ การประกอบกัน (composition) การสืบทอด (inheritance) และอินเทอร์เฟส (interface) บทนี้จะอธิบายความสัมพันธ์สามประเภทนี้

## 3.1 การประกอบกัน

ในการโปรแกรมเชิงวัตถุ ประเภทความสัมพันธ์ระหว่างคลาสหรืออ๊อบเจ็กต์ที่สำคัญที่สุดและใช้บ่อยที่สุด คือ การประกอบกัน หรือ composition ซึ่งเป็นการประกอบกันของคลาสหรืออ๊อบเจ็กต์ กล่าวคือ เป็นการนำอ๊อบเจ็กต์ของคลาสอื่นมาใช้งานเป็นตัวแปรประจำอ๊อบเจ็กต์ของคลาสเรานั่นเอง หากพิจารณารหัสคำสั่งทั้งหมดที่เรียนมาในหนังสือเล่มนี้ จะเห็นว่า ตัวแปรประจำอ๊อบเจ็กต์จะเป็นประเภทข้อมูลพื้นฐาน (primitive type) ทั้งหมด โดยเราเรียกตัวแปรประจำอ๊อบเจ็กต์ที่เป็นประเภทข้อมูลพื้นฐานนี้ว่าเป็นคุณลักษณะ (attribute หรือ property) หากเราประกาศตัวแปรประจำอ๊อบเจ็กต์ให้เป็นประเภทคลาส จะเกิดเป็นความสัมพันธ์แบบ composition นอกจากนั้น เราเรียกความสัมพันธ์แบบ composition นี้ว่าเป็นความสัมพันธ์แบบการมีหรือ “Has-A” เนื่องจากตรงกับการโปรแกรมที่เราให้อ๊อบเจ็กต์หนึ่งมีอ๊อบเจ็กต์อื่นเป็นตัวแปรประจำอ๊อบเจ็กต์

เราจะมาเรียนรู้การนำคลาสมาประกอบกัน (composition) ผ่านตัวอย่าง โดยจะเริ่มจากคลาสที่ยังไม่มี composition แล้วค่อยจัดแบ่งคลาสออกเป็นสองคลาสและประกอบกันด้วย composition

## คลาสที่ยังไม่มี composition

พิจารณาเวกเตอร์ (Vector) ซึ่งเป็นปริมาณทางคณิตศาสตร์ที่มีทั้งขนาดและทิศทาง สามารถแสดงด้วยคู่อันดับเริ่มต้นและคู่อันดับสิ้นสุด รหัสคำสั่งต่อไปนี้แสดงคลาส `Vector` ซึ่งประกอบด้วย

- ตัวแปรประจำอ๊อบเจ็กต์ มี 4 ค่า คือ `startX`, `startY`, `endX`, `endY` โดยสองค่าแรกเป็นคู่อันดับเริ่มต้น ($x_{start}$,$y_{start}$) และคู่อันดับสิ้นสุด ($x_{end}$,$y_{end}$)
- คอนสตรักเตอร์แบบปริยายที่กำหนดค่า 0 ให้กับตัวแปรประจำอ๊อบเจ็กต์ และคอนสตรักเตอร์ที่รับพารามิเตอร์มากำหนดค่าเริ่มต้นให้กับตัวแปรประจำอ๊อบเจ็กต์ทั้งสี่ค่า
- เมทอดประจำอ๊อบเจ็กต์ `add` เพื่อบวกเวกเตอร์สองอันเข้าด้วยกัน โดยเป็นการบวกค่า x แต่ละค่าเข้าด้วยกัน และบวกค่า y แต่ละค่าเข้าด้วยกัน และเมทอดประจำอ๊อบเจ็กต์
- เมทอดประจำอ๊อบเจ็กต์ `toString` เพื่อคืนค่าสตริงที่แสดงถึงเวกเตอร์นี้ ซึ่งเป็นสตริงคู่อันดับเริ่มต้นชี้ลูกศรไปที่คู่อันดับสิ้นสุด

```
1  class Vector {
2
3      private double startX, startY;
4      private double endX, endY;
5
6      public Vector() {
7          startX = 0;
8          startY = 0;
9          endX = 0;
10         endY = 0;
11     }
12
13     public Vector(double sx, double sy, double ex, double ey) {
14         startX = sx;
15         startY = sy;
16         endX = ex;
17         endY = ey;
18     }
19
20     public Vector add(Vector v2) {
21         Vector v3 = new Vector();
22         v3.startX = startX + v2.startX;
```

```
23          v3.startY = startY + v2.startY;
24          v3.endX = endX + v2.endX;
25          v3.endY = endY + v2.endY;
26          return v3;
27      }
28
29      public String toString() {
30          String s = "(" + startX + "," + startY + ")" +
31                     " ---> " +
32                     "(" + endX + "," + endY + ")";
33          return s;
34      }
35  }
```

สังเกตว่าคลาสเวกเตอร์ประกอบด้วยตัวแปรประจำอ๊อบเจ็กต์ซึ่งมี 4 ค่า ทั้งสี่เป็นจำนวนทศนิยม โดยสองค่าแรกเป็นคู่อันดับเริ่มต้น และสองค่าหลังเป็นคู่อันดับสิ้นสุด ในเมื่อเราสามารถมองได้ว่า เวกเตอร์ประกอบด้วยคู่อันดับสองคู่อยู่ภายใน ดังนั้น เราจึงสามารถมองคู่อันดับทั้งสองคู่นี้เป็นประเภทข้อมูลใหม่ ที่รวบรวมข้อมูล x และ y เอาไว้ จึงควรสร้างคลาสคู่อันดับ (Point) ขึ้นมา แล้วให้คลาสเวกเตอร์เก็บอ๊อบเจ็กต์ของคู่อันดับไว้ 2 อ๊อบเจ็กต์ แทนที่จะเก็บเป็นจำนวนทศนิยม 4 ค่า

## คลาสที่มี composition

รหัสคำสั่งต่อไปนี้แสดงคลาสคู่อันดับ (Point) ที่มีตัวแปรประจำอ๊อบเจ็กต์เป็นค่า x และ y ของคู่อันดับ มีคอนสตรักเตอร์แบบปริยายที่กำหนดค่า 0 ให้กับตัวแปรประจำอ๊อบเจ็กต์ และคอนสตรักเตอร์ที่รับพารามิเตอร์มากำหนดค่าเริ่มต้นให้กับตัวแปรประจำอ๊อบเจ็กต์ x และ y มีเมทอดประจำอ๊อบเจ็กต์ `add` เพื่อบวกคู่อันดับสองอันเข้าด้วยกัน และ `toString` เพื่อคืนค่าสตริงที่แสดงถึงคู่อันดับนี้

สังเกตว่า เมอดประจำอ๊อบเจ็กต์ทั้งสองนี้มีความคล้ายคลึงกับเมทอดประจำอ๊อบเจ็กต์ของคลาสเวกเตอร์ โดยเราได้จัดแบ่งส่วนของการทำงานที่เคยอยู่ในคลาสเวกเตอร์ก่อนหน้ามาใส่ไว้ในคลาสคู่อันดับ เพื่อแบ่งเบาภาระการคำนวณ โดยคัดเลือกการทำงานที่เกี่ยวข้องกับคู่อันดับมาไว้ให้ตรงกับบริบทการทำงาน

```
1   class Point {
2
3       private double x;
4       private double y;
5
6       public Point() {
7           x = 0;
8           y = 0;
```

```
    }

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Point add(Point other) {
        return new Point(x + other.x, y + other.y);
    }

    public string toString() {
        return "(" + x + "," + y + ")";
    }
}
```

เมื่อได้คลาสคู่อันดับแล้ว เราจะนำคลาสคู่อันดับมาประกอบกับคลาสเวกเตอร์ โดยให้คลาสเวกเตอร์มีตัวแปรประจำอ๊อบเจ็กต์เป็นประเภทของคลาสคู่อันดับ ดังรหัสคำสั่งต่อไปนี้ โดยสังเกตตัวแปรประจำอ๊อบเจ็กต์ 2 ตัว ตัวแรกเก็บอ๊อบเจ็กต์ชื่อ `start` แสดงถึงคู่อันดับเริ่มต้น และตัวที่ 2 อ๊อบเจ็กต์ชื่อ `end` แสดงถึงคู่อันดับสิ้นสุด คลาสเวกตอร์ใหม่นี้มีการนำเมทอดประจำอ๊อบเจ็กต์ของคลาสคู่อันดับมาใช้ ทั้งในเมทอด `add` และเมทอด `toString` โดยจะเห็นว่าในเมทอด `add` การบวกเวกเตอร์ คือ การนำคู่อันดับเริ่มต้นของสองเวกเตอร์มาบวกกัน และการนำคู่อันดับสิ้นสุดของสองเวกเตอร์มาบวกกัน เมทอด `toString` ก็เป็นการเรียกเมทอด `toString` ของอ๊อบเจ็กต์คู่อันดับเริ่มต้นและคู่อันดับสิ้นสุด แล้วนำผลมาเชื่อมต่อกันด้วยเครื่องหมายลูกศร

```
class Vector {

    private Point start;
    private Point end;

    public Vector(Point start, Point end) {
        this.start = start;
        this.end = end;
    }

    public Vector add(Vector other) {
        Point newStart = start.add(other.start);
        Point newSnd = end.add(other.end);
        return new Vector(newStart, newEnd);
   }

```

```
17     public string toString() {
18         return start.toString() + "-->" + end.toString();
19      }
20  }
```

การโปรแกรมในลักษณะนี้ ทำให้เข้าใจความหมายของรหัสคำสั่งในคลาสเวกเตอร์ได้ง่ายขึ้น โดยไม่จำเป็นต้องเข้าใจรายละเอียดการทำงานภายในคลาสคู่อันดับ ลองจินตนาการว่า หากคลาสคู่อันดับมีการเปลี่ยนแปลงการทำงานภายใน เช่น เพิ่มตัวแปร z ขึ้นมากลายเป็น 3 มิติ คลาสเวกเตอร์ก็ยังสามารถบวกและคืนค่าเป็นสตริงได้ โดยไม่ต้องเปลี่ยนแปลงการทำงานของเมทอดนี้ในคลาสเวกเตอร์ ทำให้โปรแกรมยืดหยุ่น การนำคลาสมาประกอบกันแบบ composition จึงถือว่ามีความสำคัญมากในการโปรแกรมเชิงวัตถุ

เมื่อประกาศคลาสเวกเตอร์ให้มีอ๊อบเจ็กต์ของคลาสคู่อันดับเป็นตัวแปรประจำอ๊อบเจ็กต์แล้ว รหัสคำสั่งต่อไปนี้เป็นการสร้างอ๊อบเจ็กต์ของคลาสคู่อันดับ แล้วนำอ๊อบเจ็กต์นี้ไปประกอบใส่ให้กับอ๊อบเจ็กต์ของคลาสเวกเตอร์

```
1   class VectorMain {
2
3       public static void main(String[] args) {
4
5           Point p1 = new Point(1,1)
6           Point p2 = new Point(2,2);
7
8           Point pAdd = p1.add(p2);
9
10          System.out.println("p1: " + p1);
11          System.out.println("p2: " + p2);
12          System.out.println("pAdd: " + pAdd);
13          System.out.println("-----------");
14
15          Point p3 = new Point(3,3);
16          Point p4 = new Point(4,4);
17
18          Vector v1 = new Vector(p1,p2);
19          Vector v2 = new Vector(p3,p4);
20
21          Vector vAdd = v1.add(v2);
22
23          System.out.println("p1: " + p1);
24          System.out.println("p2: " + p2);
25          System.out.println("pAdd: " + pAdd);
26      }
27  }
```

สังเกตรหัสคำสั่งบรรทัดต่อไปนี้

- บรรทัดที่ 5-8 สร้างอ๊อบเจ็กต์ของคลาสคู่อันดับ แล้วเรียกใช้เมทอด `add`
- บรรทัดที่ 15-16 สร้างอ๊อบเจ็กต์ของคลาสคู่อันดับเพิ่มเติมอีก 2 อ๊อบเจ็กต์
- บรรทัดที่ 18-19 เป็นการนำอ๊อบเจ็กต์ของคลาสคู่อันดับและคลาสเวกเตอร์มาประกอบกัน โดยส่งอ๊อบเจ็กต์ของคลาสคู่อันดับไปให้คลาสเวกเตอร์
- บรรทัดที่ 21 เรียกใช้เมทอด `add` ของคลาสเวกเตอร์
- บรรทัดที่ 10-12, 23-25 พิมพ์ค่าของแต่ละอ๊อบเจ็กต์ออกทางหน้าจอ

เมื่อรันเมทอด `main` จะได้ผลการรันดังต่อไปนี้

```
p1: (1,1)
p2: (2,2)
pAdd: (3,3)
-----------
v1: (1,1)-->(2,2)
v2: (3,3)-->(4,4)
vAdd: (4,4)-->(6,6)
```

## ตัวอย่างเพิ่มเติม

รหัสคำสั่งต่อไปนี้เป็นตัวอย่าง composition เพิ่มเติม พิจารณาโปรแกรมธนาคารที่นอกเหนือจากการฝากและถอนเงินแล้ว ยังสามารถโอนเงินจากบัญชีธนาคารหนึ่งไปยังอีกบัญชีธนาคารหนึ่งได้ และสามารถเปิดบัญชีกับธนาคารได้ โปรแกรมจะใช้คลาสบัญชีธนาคารเดิมและสร้างคลาสธนาคาร (`Bank`) เพิ่มเติม โดยคลาสธนาคารจะประกอบด้วยบัญชีธนาคารหลายบัญชี

รหัสคำสั่งต่อไปนี้เป็นคลาสบัญชีธนาคารเดิม

```
1  class BankAccount {
2
3      private String name;
4      private double balance;
5
6      public BankAccount(String name, double balance) {
7          this.name = name;
8          this.balance = balance;
9          if (balance < 0)
10             this.balance = 0;
11     }
12
```

```
13      public BankAccount(String name) {
14          this(name, 0);
15      }
16
17      void deposit(double amount) {
18          if (amount > 0)
19              balance += amount;
20      }
21
22      void withdraw(double amount) {
23          if (amount > 0 && amount < balance)
24              balance -= amount;
25      }
26
27      String getName() {
28          return name;
29      }
30
31      double getBalance() {
32          return balance;
33      }
34
35      void setName(String name) {
36          this.name = name;
37      }
38  }
```

รหัสคำสั่งต่อไปนี้เป็นคลาสธนาคาร ซึ่งมีตัวแปรประจำอ๊อบเจ็กต์ 2 ตัว คือ ตัวแปร `accounts` เป็น ArrayList ที่เก็บอ๊อบเจ็กต์ของคลาสบัญชีธนาคารหลายอ๊อบเจ็กต์ และตัวแปร `name` เก็บชื่อธนาคาร

คอนสตรักเตอร์รับชื่อธนาคารเป็นพารามิเตอร์และสร้างอ๊อบเจ็กต์ ArrayList เตรียมไว้รับอ๊อบเจ็กต์บัญชีธนาคาร เมทอด `openAccount` เป็นการเปิดบัญชีธนาคาร ซึ่งก็คือการเพิ่มอ๊อบเจ็กต์บัญชีธนาคารเข้าไปใน ArrayList ทำให้เกิด composition และเมทอด `transfer` เป็นการโอนเงินระหว่างบัญชี

```
1   class Bank {
2
3       private String name;
4       ArrayLIst<BankAccount> accounts;
5
6       public Bank(String name) {
7           this.name = name;
8           this.accounts = new ArrayList<>();
9       }
10
```

```
11      public void openAccount(BankAccount account) {
12          accounts.add(account);
13      }
14
15      public void transfer(BankAccount from, BankAccount to, double amount) {
16          from.withdraw(amount);
17          to.deposit(amount);
18      }
19  }
```

คล้ายกับกรณีคลาสคู่อันดับและคลาสเวกเตอร์ เราได้จัดแบ่งรหัสคำสั่งไว้ในแต่ละคลาสที่นำมาประกอบกัน โดยกระจายให้เมทอดในแต่ละคลาสทำหน้าที่ตรงกับบริบทการใช้งานของคลาสนั้น ๆ ในที่นี้ การฝากและการถอนจะสามารถทำได้กับบัญชีธนาคารโดยตรง จึงมีเมทอดฝากและถอนที่คลาสบัญชีธนาคาร ส่วนการโอนซึ่งเกี่ยวข้องกับอ๊อบเจ็กต์บัญชีธนาคารสองอ๊อบเจ็กต์ จึงเป็นหน้าที่ของคลาสธนาคาร เนื่องจากได้เก็บอ๊อบเจ็กต์ของทั้งสองบัญชีธนาคารเอาไว้

เมทอด `main()` ต่อไปนี้แสดงการใช้งานคลาสบัญชีธนาคารและคลาสธนาคาร รวมถึงการนำคลาสมาประกอบกัน

```
1   class BankMain {
2
3       public static void main(String[] args) {
4           Bank myBank = new Bank("My Bank");
5
6           BankAccount kwan = new BankAccount("Kwan", 500);
7           myBank.openAccount(kwan);
8
9           BankAccount ploy = new BankAccount("Ploy", 1000);
10          myBank.openAccount(ploy);
11          myBank.transfer(kwan, ploy, 100);
12
13          System.out.println("Kwan: " + kwan.getBalance());
14          System.out.println("Ploy: " + ploy.getBalance());
15      }
16  }
```

โดยผลการรันได้ดังนี้

```
Kwan: 400
Ploy: 1100
```

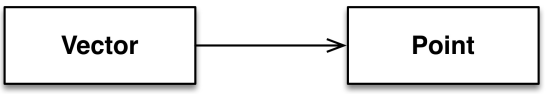
## 3.2 แผนภาพ UML

เมื่อโปรแกรมมีจำนวนคลาสมากขึ้น การทำความเข้าใจโปรแกรมจากรหัสคำสั่งอาจทำได้ยากและใช้เวลานาน การวาดรูปสัญลักษณ์ที่สามารถใช้แทนคลาส รวมถึง ตัวแปรประจำอ๊อบเจ็กต์ เมทอดประจำอ๊อบเจ็กต์ คอนสตรักเตอร์ และความสัมพันธ์ระหว่างคลาสต่าง ๆ จะช่วยให้สามารถทำความเข้าใจความหมายและการทำงานของโปรแกรมได้เร็วขึ้น

UML ย่อมาจาก Unified Modeling Language เป็นแผนภาพมาตรฐานที่มีสัญลักษณ์แสดงถึงการโปรแกรมเชิงวัตถุ แต่ละสัญลักษณ์มีความหมายชัดเจน โดยในรูปที่ 3.1 เป็นสัญลักษณ์กรอบสี่เหลี่ยมที่แสดงถึงคลาสคู่อันดับ เรียกว่าแผนภาพคลาส UML (UML class diagram)


รูปที่ 3.1 แผนภาพ UML แสดงคลาสคู่อันดับ

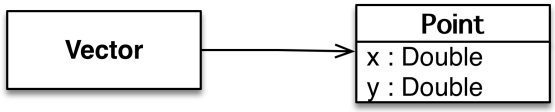
แผนภาพในรูปที่ 3.2 แสดงความสัมพันธ์ระหว่างคลาสเวกเตอร์และคลาสคู่อันดับ เครื่องหมายลูกศรแสดงว่า คลาสเวกเตอร์มีอ๊อบเจ็กต์คู่อันดับเป็นตัวแปรประจำอ๊อบเจ็กต์อยู่ภายใน




รูปที่ 3.2 แผนภาพ UML แสดงความสัมพันธ์ระหว่างคลาสเวกเตอร์และคลาสคู่อันดับ

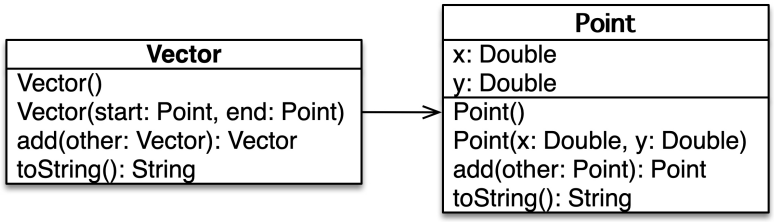
เราสามารถเพิ่มรายละเอียดเข้าไปในแผนภาพ UML ได้ โดยใส่ตัวแปรประจำอ๊อบเจ็กต์ที่เป็นคุณลักษณะ (attribute) ประเภทข้อมูลพื้นฐานได้ โดยในรูปที่ 3.3 เพิ่มกรอบสี่เหลี่ยมเพื่อใส่คุณลักษณะในคลาสคู่อันดับ สังเกตว่า ในแผนภาพ UML ประเภทข้อมูลจะอยู่หลังชื่อตัวแปร โดยมีเครื่องหมาย : คั่น

นอกจากนั้น สังเกตด้วยว่าคลาสเวกเตอร์ไม่มีตัวแปรประจำอ๊อบเจ็กต์ที่เป็นคุณลักษณะ (attribute) จึงไม่มีกรอบใส่คุณลักษณะ ส่วนตัวแปรประจำอ๊อบเจ็กต์ `start` และ `end` ที่มีประเภทเป็นคลาสคู่อันดับนั้น เราถือว่าเป็น composition ซึ่งเป็นความสัมพันธ์ระหว่างคลาส ซึ่งแสดงด้วยสัญลักษณ์เส้นและลูกศรอยู่แล้ว จึงไม่ต้องใส่ตัวแปรประจำอ๊อบเจ็กต์ที่เป็น composition ในกรอบคุณลักษณะ ดังนั้น เมื่อเราเห็นลูกศรในลักษณะนี้ให้ระลึกว่า คลาสมีอ๊อบเจ็กต์ของอีกคลาสเป็นตัวแปรประจำอ๊อบเจ็กต์



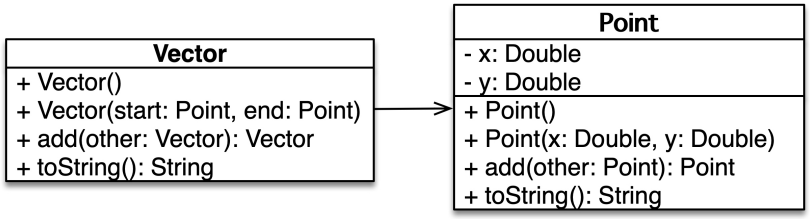

รูปที่ 3.3 แผนภาพ UML แสดงคุณลักษณะในคลาสคู่อันดับ

เพื่อให้ละเอียดขึ้น เราสามารถเพิ่มข้อมูลเมทอดประจำอ๊อบเจ็กต์และคอนสตรักเตอร์เข้าไปได้ด้วยดังรูปที่ 3.4 สังเกตประเภทข้อมูลของพารามิเตอร์ ซึ่งอยู่หลังชื่อพารามิเตอร์ และสังเกตประเภทผลลัพธ์ของเมทอดประจำอ๊อบเจ็กต์ จะอยู่หลังการประกาศเมทอดเช่นกัน



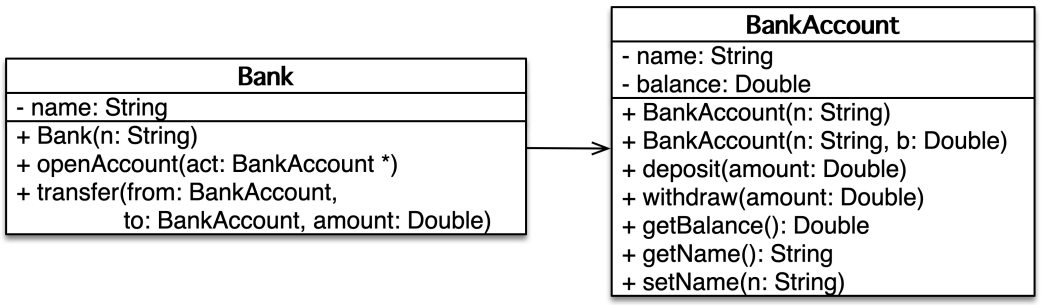

รูปที่ 3.4 แผนภาพ UML แสดงรายละเอียดในคลาสเวกเตอร์และคลาสคู่อันดับ

นอกจากนั้น เรายังสามารถแสดงการควบคุมการเข้าถึงตัวแปรประจำอ๊อบเจ็กต์ เมทอดประจำอ๊อบเจ็กต์ และคอนสตรักเตอร์ได้ ด้วยการใส่เครื่องหมายบวกและลบก่อนชื่อตัวแปรประจำอ๊อบเจ็กต์ เมทอดประจำอ๊อบเจ็กต์ และคอนสตรักเตอร์ โดยเครื่องหมายบวกหมายถึง public และเครื่องหมายลบหมายถึง private ดังแสดงในรูปที่ 3.5




รูปที่ 3.5 แผนภาพ UML แสดงรายละเอียดในคลาสเวกเตอร์และคลาสคู่อันดับ

สำหรับคลาสธนาคารและคลาสบัญชีธนาคาร เราสามารถวาดแผนภาพคลาส UML ได้ดังรูปที่ 3.6




รูปที่ 3.6 แผนภาพ UML แสดงความสัมพันธ์ของคลาสธนาคารและคลาสบัญชีธนาคาร

## 3.3 การสืบทอด

การสืบทอด (inheritance) เป็นความสัมพันธ์ระหว่างคลาสประเภทหนึ่ง มีประโยชน์ช่วยให้สามารถเรียกใช้เมทอดของคลาสอื่นที่มีอยู่แล้วประหนึ่งเป็นเมทอดของคลาสตนเอง แทนการเขียนโปรแกรมซ้ำขึ้นมา และยังสามารถปรับแต่งเมทอดนั้นให้เข้ากับสถานการณ์ใหม่ได้ด้วย ส่วนใหญ่จะใช้เมื่อโปรแกรมมีการทำงานหลายแบบคล้ายกัน และแต่ละแบบมีการทำงานบางอย่างที่เหมือนกัน และบางอย่างที่เฉพาะเจาะจงสำหรับแบบนั้น ๆ

เช่น หากธนาคารมีบัญชี 3 ชนิด คือ (1) บัญชีธนาคารแบบปกติ (2) บัญชีธนาคารแบบออมทรัพย์ (3) บัญชีธนาคารแบบมีประกันชีวิต โดยบัญชีทั้งสามแบบมีการทำงานที่เหมือนกันคือ สามารถฝากและถอนได้ แต่บัญชีธนาคารแบบออมทรัพย์จะพิเศษตรงที่ผู้ใช้จะได้ดอกเบี้ยด้วย ส่วนบัญชีธนาคารแบบมีประกันชีวิตจะพิเศษตรงที่ผู้ใช้จะได้ประกันชีวิตด้วย แต่มีข้อกำหนดว่า ถอนได้จำนวนจำกัดต่อเดือน

หากเราสร้างคลาสมา 3 คลาสตามชนิดของบัญชีธนาคาร คือ (1) คลาสบัญชีธนาคารแบบปกติ (2) คลาสบัญชีธนาคารแบบออมทรัพย์ (3) คลาสบัญชีธนาคารแบบมีประกันชีวิต ทั้งสามคลาสจะมีเมทอดการฝากและการถอนเหมือนกัน และมีเมทอดอื่นที่เฉพาะเจาะจงของชนิดบัญชีธนาคารนั้น ๆ ซึ่งจะทำให้รหัสคำสั่งส่วนการฝากและการถอนต้องเขียนซ้ำกันถึง 3 ครั้ง ส่งผลให้รหัสคำสั่งไม่กระชับ และในกรณีที่ต้องแก้ไขเมทอดเหล่านี้ เราอาจแก้ไม่ครบถ้วนทุกที่ ทำให้เกิดข้อผิดพลาดได้ง่าย

การสืบทอดจะช่วยลดความซ้ำซ้อนของรหัสคำสั่งเหล่านี้ โดยแบ่งประเภทคลาสเป็น “superclass” และ “subclass” โดย

- **superclass** จะเป็นคลาสที่มีความทั่วไป ไม่เฉพาะเจาะจง มีเมทอดที่หลายคลาสต้องมีเหมือนกัน
- **subclass** จะเป็นคลาสที่มีความเฉพาะเจาะจง มีเมทอดที่เฉพาะกับคลาสนี้เท่านั้น

อาจมองได้ว่า superclass เป็นคลาสกลางที่รวบรวมเมทอดที่หลายคลาสใช้ร่วมกันมาไว้ด้วยกัน และ sub-

class เป็นคลาสที่ต่อยอดมาจากคลาสกลาง มีการปรับหรือเพิ่มเติมเมทอดที่เฉพาะจงเข้ามา นอกจากนั้นเราจะเรียกกลุ่มคลาสที่ประกอบด้วย superclass หนึ่ง ๆ และ subclass ของ superclass นั้นทั้งหมดว่า **class hierarchy** หรือโครงสร้างลำดับชั้นของคลาส ภาษาโปรแกรมเชิงวัตถุอื่นอาจเรียก superclass ว่า parent class หรือ base class และเรียก subclass ว่า child class หรือ derived class

สำหรับตัวอย่างบัญชีธนาคารสามชนิด คลาสบัญชีธนาคารแบบปกติสามารถเป็น “superclass” เนื่องจากมีความทั่วไป มีเมทอดการฝากและถอนซึ่งบัญชีธนาคารทุกชนิดต้องมีเหมือนกัน ส่วนคลาสบัญชีธนาคารแบบออมทรัพย์และคลาสบัญชีธนาคารแบบมีประกันชีวิตจะเป็น “subclass” เนื่องจากบัญชีทั้งสองชนิดมีความเฉพาะเจาะจง กล่าวคือ มีเมทอดที่คลาสอื่นไม่มี แบบออมทรัพย์จะมีเมทอดการให้ดอกเบี้ยแต่อีกสองคลาสไม่มี แบบมีประกันชีวิตก็จะมีเมทอดการประกันแต่อีกสองคลาสไม่มี

กลไกการสืบทอดในการโปรแกรมเชิงวัตถุ จะให้คลาสที่เป็น subclass ได้รับการ “สืบทอด” จากคลาสที่เป็น superclass โดยการสืบทอดจะทำให้ subclass ได้รับตัวแปรและเมทอดประจำอ๊อบเจ็กต์ทั้งหมดจาก superclass ทำให้สามารถเรียกใช้เมทอดประจำอ๊อบเจ็กต์ของ superclass ได้เสมือนเป็นเมทอดของตนเองโดยไม่ต้องนิยามเมทอดใหม่ให้ซ้ำซ้อนกัน สำหรับตัวแปรประจำอ๊อบเจ็กต์ที่ subclass ได้รับนั้น subclass อาจจะเข้าถึงได้โดยตรงหรือไม่ก็ได้ ขึ้นอยู่กับ access specifier ที่ superclass ได้กำหนดไว้ หัวข้อย่อยจะอธิบายรายละเอียดเพิ่มเติมในประเด็นนี้ อย่างไรก็ตาม subclass จะไม่ได้รับคอนสตรักเตอร์ของ superclass มาด้วย จึงต้องสร้างขึ้นมาเอง แต่สามารถเรียกใช้คอนสตรักเตอร์ของ superclass ได้

เพื่อให้ subclass มีความเฉพาะเจาะจงและตรงกับการใช้งาน subclass สามารถเพิ่มตัวแปรและเมทอดประจำอ๊อบเจ็กต์ใหม่ที่เฉพาะสำหรับคลาสนั้น ๆ ได้ รวมทั้งสามารถปรับแต่งเมทอดประจำอ๊อบเจ็กต์จาก superclass ได้ด้วย การปรับแต่งเมทอดประจำอ๊อบเจ็กต์เรียกว่าการ override หัวข้อย่อยจะอธิบายรายละเอียดเพิ่มเติม

## ตัวอย่างการสืบทอด

เราจะมาเรียนรู้การสืบทอดให้ลึกซึ้งขึ้นจากตัวอย่างง่าย ๆ สมมติว่า เราต้องการเขียนโปรแกรมเกมสัตว์เลี้ยง โดยมีสัตว์เลี้ยง 3 ชนิด คือ สุนัข แมว และนก แต่ละชนิดสามารถกินอาหารและนอนได้เหมือนกัน และสามารถเรียกร้องความสนใจด้วยเสียงได้ด้วย แต่ละมีเสียงร้องที่แตกต่างกันไป

### โค้ดซ้ำซ้อนเมื่อไม่ใช้การสืบทอด

หากเราเขียนโปรแกรมโดยสร้าง 3 คลาส คือ คลาส Dog คลาส Cat และคลาส Bird เราจะได้คลาสดังรหัสคำสั่งต่อไปนี้ ซึ่งจะเห็นว่ามีรหัสคำสั่งที่ซ้ำกันอยู่หลายจุดมากทั้งตัวแปรและเมทอดประจำอ๊อบเจ็กต์

```
class Dog {
    private String name;
    private int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }

    public void eat(string food) {
        System.out.println(name + " is eating " + food);
    }
    public void sleep() {
        System.out.println(name + " is sleeping...");
    }
    public void woof() {
        System.out.println(name + " is woofing");
    }
}
```

```
class Cat {
    private String name;
    private int age;

    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }

    public void eat(string food) {
        System.out.println(name + " is eating " + food);
    }
    public void sleep() {
        System.out.println(name + " is sleeping...");
    }
    public void meow() {
        System.out.println(name + " is meowing");
    }
}
```

```
class Bird {
    private String name;
    private int age;

    public Bird(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }

    public void eat(string food) {
        System.out.println(name + " is eating " + food);
    }
    public void sleep() {
        System.out.println(name + " is sleeping...");
    }
    public void chirp() {
        System.out.println(name + " is chirping");
    }
}
```

จะเห็นว่า ทั้งสามคลาสมีรหัสคำสั่งที่ซ้ำกัน คือ ตัวแปรประจำอ๊อบเจ็กต์ `name` และ `age` เมทอดประจำอ๊อบเจ็กต์ `getName()`, `getAge()`, `eat()`, `sleep()` เมทอดที่ต่างกันจะมีเพียงแค่เมทอดการเรียกร้องความสนใจด้วยเสียงที่ต่างกันเท่านั้น คือ เมทอด `woof()` ของคลาส Dog เมทอด `meow()` ของคลาส Cat และเมทอด `chirp()` ของคลาส Bird

เมื่อนิยามคลาสทั้งสามแล้ว เราสามารถเรียกใช้คลาสเหล่านี้ได้ดังรหัสคำสั่งต่อไปนี้

```
public static void main(String[] args) {
    Dog toob = new Dog("Toob", 2);
    Cat maw = new Cat("Maw", 3);
    Bird nok = new Bird("Nok", 1);

    toob.eat("pork");
    toob.woof();

    maw.eat("tuna");
    maw.meow();

    nok.eat("worm");
    nok.chirp();
}
```

รหัสคำสั่งข้างต้นสร้างอ๊อบเจ็กต์ของคลาส Dog คลาส Cat และคลาส Bird มาอย่างละหนึ่งอ๊อบเจ็กต์ โดยแต่ละอ๊อบเจ็กต์จะเรียกใช้เมทอดกินและส่งเสียงร้องของตนเอง

เมื่อคอมไพล์และรัน จะได้ผลการทำงานดังนี้ โดยจะเห็นว่าสัตว์แต่ละชนิดส่งเสียงร้องที่ต่างกัน

```
Toob is eating pork
Toob is woofing
Maw is eating tuna
Maw is meowing
Nok is eating worm
Nok is chirping
```

**โค้ดไม่ซ้ำซ้อนเมื่อใช้การสืบทอด**

หากนำการสืบทอดมาใช้ จะลดความซ้ำซ้อนของรหัสคำสั่งได้ โดยเราสามารถนิยาม superclass ให้รวบรวมตัวแปรประจำอ๊อบเจ็กต์ที่ทั้งสามคลาสมีเหมือนกัน คือ ชื่อและอายุ และรวบรวมเมทอดประจำอ๊อบเจ็กต์ที่ทั้งสามคลาสมีเหมือนกัน คือ การนอนและการกินอาหาร ไว้ด้วยกัน การสืบทอดจะทำให้ subclass ไม่ต้องเขียนตัวแปรประจำอ๊อบเจ็กต์และเมทอดประจำอ๊อบเจ็กต์เหล่านี้ซ้ำอีกรอบ แต่สามารถเรียกใช้ได้ประหนึ่งเป็นตัวแปรประจำอ๊อบเจ็กต์และเมทอดประจำอ๊อบเจ็กต์ของตนเอง เราเรียกการใช้ตัวแปรประจำอ๊อบเจ็กต์และเมทอดประจำอ๊อบเจ็กต์จากคลาสอื่นในลักษณะนี้ว่า การรียูสรหัสคำสั่ง (code reuse) ซึ่งจะทำให้การพัฒนาซอฟต์แวร์มีประสิทธิภาพมากขึ้น ลดเวลาพัฒนา และลดข้อผิดพลาดได้ด้วย

นอกจากเมทอดการนอนและการกินอาหารที่เหมือนกันแล้ว ทั้งสามคลาสยังมีเมทอดการเรียกร้องความสนใจที่คล้ายกัน ต่างกันแค่เสียงร้องเท่านั้น ในกรณีนี้ เราควรสร้างเมทอดการเรียกร้องความสนใจไว้ที่ superclass ด้วย แต่ให้ subclass สามารถปรับแต่งได้ตามต้องการ ดังนั้น เราจึงต้องตั้งชื่อเมทอดและนิยามเมทอดนี้ใหม่ เพื่อให้มีความเป็นกลาง สามารถปรับใช้ได้กับทุก subclass โดยในที่นี้เราจะตั้งชื่อเมทอดการเรียกร้องความสนใจว่า `makeNoise()` การสร้างเมทอดไว้ที่ superclass ให้มีความเป็นกลางและไม่เฉพาะเจาะจงในลักษณะนี้ จะช่วยให้เกิด polymorphism ได้ ดังจะอธิบายในหัวข้อถัดไป

สำหรับตัวอย่างสัตว์เลี้ยง เราจะสร้าง superclass ชื่อ Animal ที่รวบรวมมตัวแปรประจำอ๊อบเจ็กต์และเมทอดประจำอ๊อบเจ็กต์ที่ทั้งสามคลาสมีเหมือนกันไว้ด้วยกัน ดังรหัสคำสั่งต่อไปนี้

```
1  class Animal {
2
3      private string name;
4      private int age;
5
6      public Animal(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }

    public void eat(string food) {
        System.out.println(name + " is eating " + food);
    }
    public void sleep() {
        System.out.println(name + " is sleeping...");
    }
    public void chirp() {
        System.out.println(name + " is making noise");
    }
}
```

และให้คลาส Dog คลาส Cat และคลาส Bird เป็น subclass โดยสืบทอดจากคลาส Animal การประกาศ subclass จะใช้รูปแบบดังนี้

```
class SubclassName extends SuperclassName { ... }
```

รหัสคำสั่งต่อไปนี้ประกาศให้คลาสสัตว์เลี้ยงทั้งสามเป็น subclass ของคลาส Animal และมีการปรับแต่งเมทอด `makeNoise()` ให้เฉพาะเจาะจงสำหรับแต่ละคลาส

```
class Dog extends Animal {

    public Dog(String name, int age) {
        super(name, age);
    }

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

```
class Cat extends Animal {

    public Cat(String name, int age) {
        super(name, age);
    }

    public void makeNoise() {
```

```
8         System.out.println(name + " is meowing");
9     }
10 }
```

```
1  class Bird extends Animal {
2
3      public Bird(String name, int age) {
4          super(name, age);
5      }
6
7      public void makeNoise() {
8          System.out.println(name + " is chirping");
9      }
10 }
```

เนื่องจากเราเปลี่ยนให้เมทอดการส่งเสียงร้องของทุกคลาสใช้ชื่อเมทอดเดียวกัน จึงต้องปรับเมทอด `main()` ให้ทุกอ๊อบเจ็กต์เรียกเมทอด `makeNoise()` แทนเมทอดเดิมที่เฉพาะเจาะจง ดังรหัสคำสั่งต่อไปนี้

```
1  public static void main(String[] args) {
2      Dog toob = new Dog("Toob", 2);
3      Cat maw = new Cat("Maw", 3);
4      Bird nok = new Bird("Nok", 1);
5
6      toob.eat("pork");
7      toob.makeNoise();
8
9      maw.eat("tuna");
10     maw.makeNoise();
11
12     nok.eat("worm");
13     nok.makeNoise();
14 }
```

เมื่อคอมไพล์และรัน จะได้ผลจะให้ผลเหมือนเดิม โดยสัตว์แต่ละชนิดจะส่งเสียงร้องที่ต่างกัน

```
Toob is eating pork
Toob is woofing
Maw is eating tuna
Maw is meowing
Nok is eating worm
Nok is chirping
```

ให้สังเกตการนิยาม subclass ของคลาสสัตว์เลี้ยงทั้งสามดังนี้

- ไม่ต้องประกาศเมทอดการนอนและการกินอาหาร แต่สามารถเรียกใช้งานได้เสมือนเป็นเมทอดของตนเอง เนื่องจากได้รับทอดเมทอดทั้งสองมาจาก superclass Animal
- ไม่ต้องประกาศตัวแปรประจำอ๊อบเจ็กต์ แต่มีข้อมูลเหล่านี้อยู่ภายใน
- การเข้าถึงตัวแปรประจำอ๊อบเจ็กต์จะขึ้นอยู่กับ access specifier ของตัวแปรประจำอ๊อบเจ็กต์ที่ superclass ได้ประกาศไว้ หากประกาศเป็น public แล้ว subclass จะเข้าถึงโดยตรงได้ แต่หากเป็น private แล้ว subclass จะเข้าถึงโดยตรงไม่ได้ ต้องเข้าถึงผ่านเมทอดประจำอ๊อบเจ็กต์ของ superclass
- เนื่องจากคลาส Animal ประกาศให้ตัวแปรประจำอ๊อบเจ็กต์ `name` เป็น private ทำให้ subclass ไม่สามารถเข้าถึงได้โดยตรง แต่ subclass จะสามารถนำค่า `name` มาใช้ได้ผ่านเมทอด `getName()` ที่ได้รับการสืบทอดมาจาก superclass
- แต่ละคลาสมีคอนสตรักเตอร์ของตนเอง เนื่องจากตัวแปรประจำอ๊อบเจ็กต์ `name` และ `age` เป็น private ทำให้ subclass ไม่สามารถกำหนดค่าตัวแปรเหล่านี้เองได้ แต่สามารถเรียกใช้คอนสตรักเตอร์ของ superclass Animal ผ่านคีย์เวิร์ด `super` สังเกตรหัสคำสั่ง `super(name, age);` ในบรรทัดที่ 4 รหัสคำสั่งนี้คือการเรียกใช้คอนสตรักเตอร์ของ superclass ที่มีการส่งค่าชื่อและอายุผ่านทางพารามิเตอร์ไปให้คอนสตรักเตอร์ Animal ด้วย
- subclass ทั้งสามปรับแต่งเมทอด `makeNoise()` ให้เฉพาะเจาะจงสำหรับแต่ละคลาส

## การปรับแต่งเมทอดประจำอ๊อบเจ็กต์

การปรับแต่งเมทอดประจำอ๊อบเจ็กต์ คือ การที่ subclass นิยามเมทอดประจำอ๊อบเจ็กต์ที่มีส่วนหัว (ชื่อเมทอดและรายการพารามิเตอร์) เหมือนกับเมทอดประจำอ๊อบเจ็กต์ของ superclass แต่มีการปรับเปลี่ยนหรือเพิ่มเติมการทำงานข้างในเมทอดให้เฉพาะเจาะจงกับความต้องการของ subclass นั้น ๆ การปรับแต่งเมทอดประจำอ๊อบเจ็กต์เรียกว่าการ override