CHAPTER07

แนวคิดเชิงอ็อปเจ็ค (Object Oriented Concept)

บทนำ

- การจะเขียนโปรแกรมเชิงอ๊อบเจกต์ได้นั้น โปรแกรมเมอร์จะต้องทำความเข้าใจในแนวคิดเชิงอ๊อบเจกต์ ให้เข้าใจเสียก่อน
- ใน **chapter** นี้มีการนำเสนอแนวคิดเชิงอ๊อบเจกต์ เพื่อนำไปสู่การเขียนโปรแกรมเชิงอ๊อบเจกต์ต่อไป

Object Oriented Concept

- หลักการพื้นฐานของ **object oriented** มีดังนี้
- 1. **Encapsulation** (การหุ้มห่อ)
- 2. **Inheritance** (การสืบทอดคุณสมบัติ)
- 3. **Polymorphism** (การพ้องรูป)

Encapsulation

- เป็นกระบวนการซ่อนรายละเอียดการทำงาน และข้อมูลไว้ภายในไม่ให้ภายนอกสามารถมองเห็นได้
- ทำให้ภายนอกไม่สามารถทำการเปลี่ยนแปลงแก้ไขข้อมูลภายในได้
- ซึ่งเป็นผลทำให้ไม่เกิดความเสียหายแก่ข้อมูล
- ข้อดีของการ **encapsulation** คือ สามารถสร้างความปลอดภัยให้แก่ข้อมูลได้ เนื่องจากข้อมูลจะถูกเข้าถึงจากผู้มีสิทธิ์เท่านั้น

Inheritance

- หลักการของ **inheritance** คือ ทำการสร้างสิ่งใหม่ขึ้นด้วยการสืบทอดหรือรับเอา **(inherit)** คุณสมบัติบางอย่างมาจากสิ่งเดิมที่มีอยู่แล้ว
- โดยการสร้างเพิ่มเติมจากสิ่งที่มีอยู่แล้วได้เลย
- ข้อดีของการ **inheritance** คือ จากการที่สามารถนำสิ่งที่เคยสร้างขึ้นแล้วนำกลับมาใช้ใหม่ **(re-use)** ได้ ทำให้ช่วยประหยัดเวลาการทำงานลง เนื่องจากไม่ต้องเสียเวลาพัฒนาใหม่หมด

Polymorphism

- **Polymorphism** เกิดจาก **poly**(หลากหลาย)+**morphology** (รูปแบบ)
- ใช้ในวงการชีววิทยา จะหมายถึงสิ่งมีชีวิตชนิดเดียวกัน แต่มีรูปแบบหลากหลาย เช่นมนุษย์ชาย มนุษย์หญิง มดนางพญา มดงาน มดทหาร ฯลฯ
- ในทางโปรแกรมคือการที่เมธอดชื่อเดียวกัน สามารถรับอาร์กิวเมนต์ที่แตกต่างกันได้หลายรูปแบบ โดยเมธอดนี้จะถูกเรียกว่า **overload method** (เมธอดถูกโอเวอร์โหลด)

Class & Object

- คลาส (class) คือต้นแบบของวัตถุ การจะสร้างวัตถุขึ้นมาอย่างหนึ่งจะต้องสร้างคลาสขึ้นมาเป็นโครงสร้างต้นแบบสำหรับวัตถุก่อนเสมอ
- วัตถุหรือออบเจ็ค (object) คือสิ่งที่ประกอบไปด้วยคุณสมบัติ 2 ประการ คือ คุณลักษณะ และพฤติกรรม
- คุณลักษณะ (attribute หรือ data) คือ สิ่งที่บ่งบอกลักษณะทั่วไปของวัตถุ

Class & Object (cont.)

- พฤติกรรม (**Behavior** หรือ **method**) คือ สิ่งที่วัตถุสามารถกระทำออกมาได้
- เราเรียก **attribute** และ **method** ว่าเป็น **member** ของคลาส

ตัวอย่าง วัตถุ หรือออบเจ็ค

คลาส “โทรศัพท์มือถือ”	
attribute	ยี่ห้อ, รุ่น, จอภาพ, รูปแบบของเสียงเรียกเข้า
method	เปิด, ปิด, ตั้งนาฬิกาปลุก, เลือกรูปแบบเสียงเรียกเข้า

ตัวอย่าง วัตถุ หรือออบเจ็ค (ต่อ)

คลาส “พนักงานบริษัท”	
attribute	รหัสพนักงาน, เงินเดือน, เวลาเข้างาน, เวลาออกงาน
Method	รูดบัตรพนักงาน, รับเงินเดือน

Modifier in java language

- modifier ในภาษาจาวา แบ่งออกเป็น 2 ประเภทใหญ่ๆ คือ
- 1. access modifier ได้แก่ private, public, protected และ package (บางทีเรียกว่า none, default หรือ friendly ก็ได้)
- 2. non-access modifier ได้แก่ final, abstract, static, native, transient, volatile, synchronized, strictfp
- access modifier ใช้กำหนดไว้หน้าคลาส แอทริบิวต์ หรือเมธอดใดๆ เพื่อกำหนดระดับการเข้าใช้งาน

การนำ access modifier มาใช้งาน

	ใช้กับคลาส	ใช้กับแอทริบิวต์	ใช้กับเมธอด
public	✓	✓	✓
protected	✗	✓	✓
package	✓	✓	✓
private	✗	✓	✓

access modifier

- จาวาแบ่งระดับของ **access modifier** ออกเป็น 4 ระดับ คือ
- 1. **public** (สาธารณะ) หากกำหนด **modifier** ให้กับคลาส แอทริบิวต์ และเมธอดใดๆ แล้ว คลาสอื่นๆ จะสามารถเข้าใช้งานคลาส แอทริบิวต์ และเมธอดนั้นได้อย่างอิสระ

access modifier (cont.)

- 2. **protected** (ถูกปกป้อง) ไม่ได้เปิดให้คลาสอื่นๆ สามารถเข้าใช้งานได้อย่างอิสระ แต่ก็ไม่ถึงกับปกปิดไม่ให้ใครเข้าใช้งานเลย ดังนี้
  - คลาสที่อยู่ในแพ็กเกจเดียวกันกับคลาสที่ถูกกำหนด **modifier** เป็น **protected** จะสามารถเรียกใช้งาน **member** ของคลาสที่ถูกกำหนดเป็น **protected** ได้
  - คลาสที่อยู่ต่างแพ็กเกจกันกับคลาสที่ถูกกำหนด **modifier** เป็น **protected**จะไม่สามารถเรียกใช้งาน **member** ของคลาสที่ถูกกำหนดเป็น **protected**
  - คลาสที่อยู่ต่างแพ็กเกจกันกับคลาสที่ถูกกำหนด **modifier** เป็น **protected** แต่มีความสัมพันธ์เป็นคลาสแม่ลูกกัน สามารถเรียกใช้งาน **member** ของคลาสที่ถูกกำหนดเป็น **protected** ได้

access modifier (cont.)

- 3. **package** (แพ็กเกจ) กรณีที่ไม่ได้กำหนด **modifier** ใดๆ เลยไว้หน้าคลาส แอทริบิวต์ หรือเมธอด จะทำให้คลาส แอทริบิวต์ หรือเมธอดนั้นมีระดับการเข้าถึงเป็น **package**
- ซึ่งหมายความว่า คลาสที่อยู่ในแพ็กเกจอื่นจะไม่สามารถเข้าใช้งานคลาส แอทริบิวต์ และเมธอดเหล่านี้ได้

access modifier (cont.)

- **4. private** (ส่วนบุคคล) หมายถึงความเป็นส่วนตัว เป็นการปิดกั้นไม่ให้คลาสอื่นเข้ามาใช้งานแอทริบิวต์ และเมธอดที่ถูกกำหนด **modifier** เป็น **private** ได้
- กล่าวคือ จะมีแต่คลาสของตัวมันเองเท่านั้นที่มีสิทธิ์ใช้งานได้

สรุปการทำงาน access modifier แต่ละแบบ

	ใช้ได้ทั้งหมด	แพ็คเกจเดียวกัน	ต่างแพ็กเกจกัน	ต่างแพ็กเกจกันแต่เป็นคลาสแม่ลูกกัน	คลาสเดียวกัน
public	✓	✓	✓	✓	✓
protected	x	✓	x	✓	✓
package	✓	✓	x	x	✓
private	x	x	x	x	✓

การประกาศคลาส

- รูปแบบการประกาศคลาส

```
[modifier] class ชื่อคลาส {
        [ส่วนการประกาศแอทริบิวต์]
        [ส่วนการประกาศเมธอด]
}
```

- กฎและข้อเสนอแนะสำหรับการตั้งชื่อคลาส คือชื่อควรเป็นคำนาม และนิยมตั้งชื่อคลาสด้วยอักษรภาษาอังกฤษขึ้นต้นด้วยตัวพิมพ์ใหญ่

การประกาศคลาส (ต่อ)

- ตัวอย่างการประกาศคลาส

```
public class Employee { }
```

```
public class MobilePhone { }
```

การประกาศแอทริบิวต์

- รูปแบบการประกาศแอทริบิวต์

```
[modifier] ชนิดข้อมูล ชื่อแอทริบิวต์;
```

- กฎและข้อแนะนำสำหรับการตั้งชื่อแอทริบิวต์
- นิยมเขียนด้วยตัวอักษรภาษาอังกฤษพิมพ์เล็กทั้งหมด
- ชื่อแอทริบิวต์ควรเป็นคำนาม

การประกาศแอทริบิวต์ (ต่อ)

- ตัวอย่างการประกาศแอทริบิวต์

```
public class Employee {
        private String name;
        private int salary;
}
```

การประกาศเมธอด

- รูปแบบการประกาศเมธอด

```
[modifier] ชนิดข้อมูล ชื่อเมธอด([argument]) {
        [รายละเอียดการทำงานของเมธอด (คำสั่งต่างๆ)]
}
```

- กฎและข้อแนะนำสำหรับการตั้งชื่อเมธอด นิยมตั้งชื่อเป็นคำกริยา ชื่อเมธอดมีหลายคำ คำแรกเป็นพิมพ์เล็กทั้งหมดคำต่อไปขึ้นต้นด้วยตัวพิมพ์ใหญ่

การประกาศเมธอด (ต่อ)

- ตัวอย่างการประกาศเมธอด

```
public class Employee {
        private String name;
        private int salary;
        public int getSalary() {
                return saraly;
        }
}
```

การประกาศตัวแปรออบเจ็ค และการสร้างออบเจ็ค

- รูปแบบการประกาศตัวแปรอ๊อบเจกต์ (object declaration)

```
ชื่อคลาส ชื่อตัวแปรอ๊อบเจกต์;
```

- การสร้างอ๊อบเจกต์ (object creation)

```
ตัวแปรอ๊อบเจกต์ = new ชื่อคลาส([argument]);
```

- อ๊อบเจกต์ บางทีก็จะถูกเรียกว่า “**instance** ของคลาส”

การประกาศตัวแปรอ๊อบเจกต์ และการสร้างอ๊อบเจกต์ (ต่อ)

- ตัวอย่างการประกาศตัวแปรอ๊อบเจกต์

```
Employee emp;
MobilePhone phone;
```

- ตัวอย่างการสร้างอ๊อบเจกต์

```
emp = new Employee();
phone = new MobilePhone();
```

คลาสแม่ (superclass) คลาสลูก (subclass)

- เมื่อคลาสหนึ่งสืบทอดคุณสมบัติ (inherit) มาจากอีกคลาสหนึ่ง เราจะเรียกคลาสที่ได้รับการสืบทอดคุณสมบัติว่า “subclass” หรือทีเรียกว่า “คลาสลูก”
- และเราจะเรียกคลาสที่สืบทอดคุณสมบัติให้อีกคลาสหนึ่งว่า “superclass” หรือที่เราเรียกว่า “คลาสแม่”

คลาสแม่ (superclass) คลาสลูก (subclass)

- สำหรับคลาสที่ได้รับการสืบทอดคุณสมบัติจะต้องใช้คีย์เวิร์ด **extends** ในการระบุว่าคลาสนั้นสืบทอดคุณสมบัติมากจากคลาใด เช่น

```
public class Manager extends  Employee { }
```

```
public class LadyBug extends  Insect { }
```

คอนสตรัคเตอร์ (Constructor)

- คอนสตรัคเตอร์ เป็นส่วนที่ประกอบด้วยคำสั่งที่จำเป็นสำหรับการกำหนดค่าเริ่มต้นให้กับแอทริบิวต์ต่างๆ ของอ๊อบเจกต์ โดยคอนสตรัคเตอร์จะถูกเรียกให้ทำงานโดยอัตโนมัติเมื่อเวลาเราสร้างอ๊อบเจกต์ จากคลาสด้วยคีย์เวริ์ด **new**
- มีรูปร่างหน้าตาคลายเมธอด แต่มีข้อแตกต่างตรงที่ชื่อคอนสตรัคเตอร์จะมีชื่อเหมือนคลาสทุกประการ
- เมธอดมีการคืนค่ากลับ แต่คอนสตรัคเตอร์ไม่มีการคืนค่ากลับ
- **default contructor** คือ คอนสตรัคเตอร์ที่**ไม่มีพารามิเตอร์**

คอนสตรัคเตอร์ (Constructor) (Cont.)

- ตัวอย่างคอนสตรัคเตอร์

```
public class Employee {
        private String name;
        private int salary;
        public Employee(){
                name = “Tom Cruise”;
                salary = 25000;
        }
}
```

this & super Keyword

- คีย์เวิร์ด **this** การใช้คีย์เวิร์ด **this** คือ เมื่อต้องการเรียกใช้งานคอนสตรัคเตอร์อื่นๆ ที่อยู่ภายในคลาสเดียวกัน
- คีย์เวิร์ด **super** เมื่อต้องการเรียกคอนสตรัคเตอร์ของคลาสบรรพบุรุษให้ทำงาน สามารถทำได้โดยใช้คีย์เวิร์ด **super** **โดยมีข้อแม้ว่าการเรียกใช้งานคอนสตรัคเตอร์ของคลาสบรรพบุรุษ จะต้องกระทำก่อนการเรียกที่บรรทัดแรกสุดของคอนสตรัคเตอร์นั้น ๆ เท่านั้น**

non-access modifier static

- **static** เป็นคีย์เวิร์ดในภาษาจาวา ใช้กำหนดหน้าเมธอด หรือตัวแปรเพื่อให้เมธอดหรือแอทริบิวต์นั้นเป็นแบบ **static**
- การกำหนดให้เมธอดเป็น **static** เรียกว่า **static method** จะทำให้เราสามารถเรียกใช้งานเมธอดนั้นโดยไม่ต้องสร้างอ๊อบเจกต์
- การกำหนดให้แอทริบิวต์เป็น **static** เรียกว่า **static attibute** จะทำให้เราสามารถเรียกใช้งานแอทริบิวต์นั้นโดยไม่ต้องสร้างอ๊อบเจกต์ขึ้นมาก่อน

non-access modifier final

- **final** เป็นคีย์เวิร์ดหนึ่งในภาษาจาวา สามารถที่จะกำหนดคีย์เวิร์ด **final** นี้ให้กับ คลาส เมธอด หรือแอทริบิวต์ก็ได้
- การกำหนด **final** ให้คลาส จะทำให้คลาสนั้นไม่สามารถมี **subclass** ได้
- การกำหนด **final** ให้เมธอด จะทำให้เมธอดนั้นไม่สามารถ **override method** นั้นได้
- การกำหนด **final** ให้แอทริบิวต์ จะทำให้แอทริบิวต์นั้นเป็นค่าคงที่ **(constant)**

non-access modifier abstract

- **abstract** เป็นคีย์เวิร์ดในภาษาจาวา สามารถกำหนดคีย์เวิร์ด **abstract** นี้ให้กับ คลาส หรือเมธอด ก็ได้
- **abstract method** คือ เมธอดว่างเปล่าที่ยังไม่ได้มีการกำหนดรายละเอียดการทำงานลงไป (ยังไม่ถูก **implement**) จะถูกกำหนดรายละเอียดลงไปภายหลัง โดยคลาสลูกที่ได้รับการสืบทอดจากคลาสของ **abstract method** เหล่านั้น

non-access modifier abstract (Cont.)

- **abstract class** หากคลาสใดก็แล้วแต่ที่ประกอบไปด้วยเมธอดที่เป็น **abstract method** เพียงเมธอดเดียว จะต้องประกาศคลาสนั้นเป็น **abstract** ด้วย
- กฎของ **abstract** หากคลาสใดสืบทอดมาจาก **abstract class** คลาสนั้นจะต้องทำการระบุเมธอดทุกเมธอดที่เป็น **abstract method** ใน **abstract class** ไว้เสมอ (ไม่กำหนดรายละเอียดก็ได้แต่จะต้องมีการเขียน **abstract method** ทุกเมธอดลงไปในคลาสนั้นด้วย)

interface

- **interface** มีหลักการคล้ายกับ **abstract class** คือ สร้างอินเทอร์เฟซขึ้นมาก็เพื่อกำหนดโครงสร้างของเมธอดที่จำเป็นใช้งานขึ้นมาแต่ยังไม่ได้กำหนดรายละเอียดการทำงานใดๆ ลงไปให้กับเมธอดนั้น **(abstract method)** เมธอดในอินเทอร์เฟซจึงเป็นเมธอดที่ว่างเปล่า ซึ่งในภายหลังจึงมีการกำหนดรายละเอียดของเมธอดเหล่านั้นลงไป โดยถูกกำหนดโดยคลาสที่เรียกใช้อินเทอร์เฟซนั้นๆ

interface (cont.)

- interface กับ abstract class แตกต่างกันอย่างไร?
- 1. เมธอดบางเมธอดใน abstract class ไม่เป็น abstract method ก็ได้ แต่เมธอดทุกเมธอดใน interface เป็น abstract method

interface (cont.)

- 2. คลาสที่จะเรียกใช้งาน **abstract method** ใน **abstract class** จะต้องสืบทอดคุณสมบัติไปจาก **abstract class** นั้น แล้วจึงทำการสร้างเมธอดของตัวเองขึ้นมาให้มีชื่อเดียวกับ **abstract method** ใน **abstract class** โดยกำหนดรายละเอียดการทำงานให้กับ **abstract method** เหล่านั้นตามต้องการ (คือจะต้องทำการ **override abstract method** ใน **abstract class** นั่นเอง) แต่คลาสที่จะเรียกใช้งานเมธอดในอินเทอร์เฟซไม่จำเป็นต้องมีความสัมพันธ์ใดๆ กับอินเทอร์เฟซทั้งสิ้น

interface (cont.)

- กฎในการใช้งานอินเทอร์เฟซ
- คลาสที่ทำการอิมพลีเมนต์ **(implements)** อินเทอร์เฟซใด จะต้องเขียนเมธอดที่มีอยู่ในอินเทอร์เฟซนั้นให้ครบทุกเมธอด คือไม่ว่าจะต้องการกำหนดรายละเอียดให้แก่เมธอด หรือไม่ต้องการกำหนดรายละเอียดให้กับเมธอด จะต้องมีการเขียนเมธอดเหล่านั้นให้ครบ ไม่เช่นนั้นจะเกิดข้อผิดพลาดขึ้นในขณะคอมไพล์โปรแกรม

การเรียกใช้งานอินเทอร์เฟซ

- 1. สร้างอินเทอร์เฟซ ซึ่งมีรูปแบบการสร้างอินเทอร์เฟซ ดังนี้

```
[modifier] interface ชื่ออินเทอร์เฟซ {
        [abstract methods ต่างๆ]
}
```

การเรียกใช้งานอินเทอร์เฟซ

- 2. การเรียกใช้งานอินเทอร์เฟซ มีรูปแบบดังนี้

```
[modifier] class ชื่อคลาส implements ชื่ออินเทอร์เฟซ {
        ระบุ abstract method ในอินเทอร์เฟซทุกเมธอด
}
```

ตัวอย่างการเรียกใช้งานอินเทอร์เฟซ

- interface Entertainer

Entertainer.java

```
1 package prepare;
2
3 public interface Entertainer {
4     public void sing();
5     public void dance();
6 }
```

interface Musician

Musician.java

```
1 package prepare;
2
3 public interface Musician {
4     public void playGuitar();
5     public void playPiano();
6 }
```

ตัวอย่างการเรียกใช้งานอินเทอร์เฟซ (ต่อ)

Singer.java

```
1  package prepare;
2
3  public class Singer implements Entertainer, Musician{
4      public void sing(){
5          System.out.println("Singer can sing");
6      }
7      public void dance(){
8          System.out.println("Singer can dance");
9      }
10     public void playGuitar(){
11         System.out.println("Singer can play guitar");
12     }
13     public void playPiano(){
14         System.out.println("Singer can play piano");
15     }
16 }
```

เรียกใช้งาน **interface**

Overloading & Overriding method

- **Overloading method** คือ เมธอดที่มีชื่อเหมือนกัน และอยู่ภายในคลาสเดียวกัน สิ่งที่แยกความแตกต่างของเมธอดที่เป็น **overload method** คือ พารามิเตอร์ (เป็นผลมาจากคุณสมบัติ **Object Oriented** คือ **polymorphism**)
- **Overriding method** คือ เมธอดของคลาสลูก **(subclass)** ที่มีชื่อเหมือนกับเมธอดของคลาสแม่ **(superclass)** (เป็นผลมาจากคุณสมบัติ **Object Oriented** คือ **inheritance**)

Package & Import

- Package ในภาษาจาวา หมายถึงสิ่งที่ใช้ในการรวบรวมคลาสที่มีความเกี่ยวข้องสัมพันธ์กันไว้เป็นกลุ่มเดียวกัน เพื่อให้ง่ายต่อการบริหารจัดการ และง่ายต่อการค้นหาคลาสสำหรับการใช้งานในครั้งต่อไป
- คุณลักษณะที่สำคัญของ package
- 1. การประกาศ package จะต้องประกาศไว้ที่บรรทัดบนสุดเท่านั้น
- 2. สามารถประกาศ package ได้เพียง 1 package ต่อ 1 ไฟล์
- 3. ถ้าไม่มีการประกาศ package ไฟล์ *.class ที่ได้จะถูกเก็บไว้ที่ไดเร็กทรอรี่ปัจจุบัน

Package & Import (cont.)

- รูปแบบการประกาศ **package**

```
package mainpackage.subpackage;
```

- ขึ้นต้นด้วยคีย์เวิร์ด **package** โดยมี **mainpackage** เป็น **package** หลัก และมี **subpackage** เป็น **package** ย่อย
- **subpackage** จะมีกี่ **subpackage** ก็ได้
- ชื่อ **package** ใช้ตัวพิมพ์เล็กเท่านั้น

Package & Import (cont.)

- **Import** คือ การนำคลาสที่มีอยู่แล้วมาใช้ในโปรแกรมที่กำลังจะสร้างใหม่ เราจะต้องใช้คำสั่ง **import** เพื่อบอกให้คอมไพเลอร์ทราบว่าจะสามารถหาคลาสที่เราต้องการใช้งานได้จาก **package** ใด
- การ **import** จะต้องเรียกใช้ก่อนการประกาศคลาส

Package & Import (cont.)

- รูปแบบการใช้คำสั่ง **import**

```
import mainpackage.subpackage.classname;
```

- **Classname** คือ ชื่อของคลาสที่ต้องการนำมาใช้ ซึ่งถูกเก็บไว้ในแพ็กเกจ **mainpackage.subpackage**
- บางครั้งถ้าต้องการใช้ คลาสใน **package** มากกว่า 1 คลาส สามารถแทน **classname** ด้วยเครื่องหมาย * **(asterisk)** ได้เช่นกัน

เอกสารอ้างอิง

อรพิน ประวัติบริสุทธิ์. คู่มือเขียนโปรแกรมด้วยภาษา Java. กรุงเทพฯ : โปรวิชั่น, 2537.