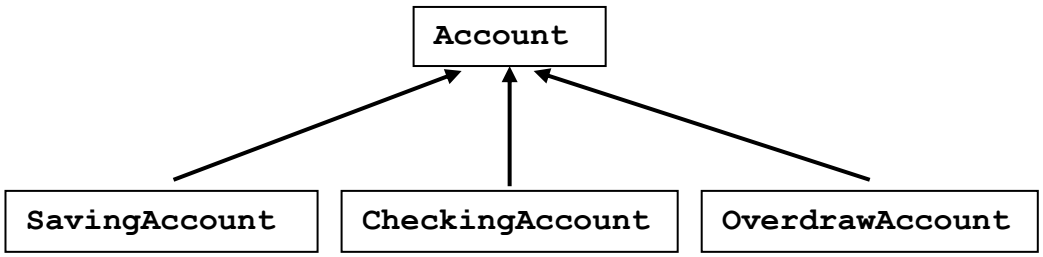
บทที่ ๕ การรับทอด (Inheritance)

การรับทอด (inheritance) เป็นหลักการที่สำคัญมากของการสร้างโปรแกรมเชิงวัตถุ หลักการนี้ช่วยเพิ่มผลิตภาพให้กับนักเขียนโปรแกรมเพราะว่าสามารถนำคลาสที่มีอยู่เดิมมาใช้ซ้ำได้โดยสะดวก ด้วยหลักการรับทอด ทำให้นักเขียนโปรแกรมสามารถเขียนคลาสใหม่โดยอาศัยคลาสอื่นที่เคยเขียนไว้เองหรือมีผู้อื่นเขียนไว้ก่อนแล้วให้เป็นประโยชน์

การรับทอดเป็นความสัมพันธ์ระหว่างคลาสที่ทำให้คลาสหนึ่งสามารถรับเอา attribute (ตัวแปร) และ behavior (เมท็อด) ของคลาสอื่นมาใช้ในคลาสของมันได้ ความสัมพันธ์แบบนี้เราแสดงด้วยแผนผังลำดับชั้น (hierarchy chart) ดังตัวอย่างต่อไปนี้



รูปที่ ๑ ผังลำดับชั้น แสดงความสัมพันธ์ระหว่างบัญชีเงินฝากธนาคารชนิดต่างๆ

บัญชีเงินฝากธนาคาร (Account) แบ่งเป็นหลายชนิดเช่น บัญชีสะสมทรัพย์ (Saving Account) บัญชีกระแสรายวันเพื่อการใช้เช็ค (Checking Account) และบัญชีกระแสรายวันที่สามารถเบิกเกินบัญชี (Overdraw Account) เป็นต้น ความสัมพันธ์แบบนี้เรียกเรียกอีกอย่างว่าเป็นความสัมพันธ์แบบ “**คือ**” (“ is – a “) เราสามารถพูดได้ว่า “บัญชีสะสมทรัพย์คือบัญชีเงินฝากธนาคาร” หรือจะพูดว่า .”บัญชีกระแสรายวันเพื่อการใช้เช็คก็คือบัญชีเงินฝากธนาคาร” ก็ได้ แต่เราจะพูดในทิศทางกลับกันไม่ได้ เช่นจะพูดว่า “บัญชีเงินฝากธนาคารคือบัญชีสะสมทรัพย์” ไม่ได้

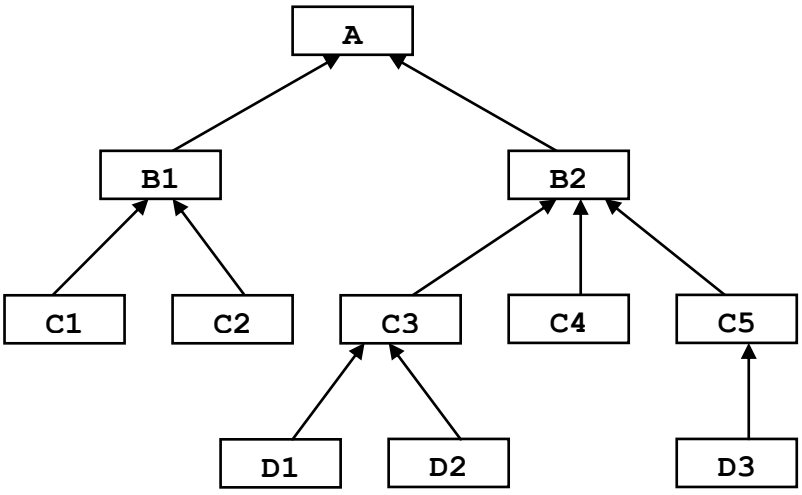
เนื่องจากบัญชีแต่ละชนิดมีคุณสมบัติบางอย่างที่แตกต่างกัน ในการสร้างโปรแกรมเชิงวัตถุนั้นเรานิยมเขียนเป็นหลายคลาสแยกกันออกไปตามชนิดของบัญชี ถึงแม้ว่าจะมีคุณสมบัติบางอย่างที่เหมือนกันก็ตาม การเขียนแยกเป็นหลายคลาสนี้ดูเหมือนทำให้โปรแกรมยุ่งยากหรือซับซ้อนมากขึ้น แต่ด้วยหลักการรับทอด ทำให้การเขียนโปรแกรมแบบนี้ไม่ยุ่งยากอย่างที่คิด ตรงกันข้าม หากรู้จักการใช้หลักการรับทอดแล้ว กลับจะทำให้รายละเอียดที่จะเขียนลงในคลาสลดน้อยลง

ในวงการเขียนโปรแกรมเชิงวัตถุ มีศัพท์ที่ใช้เรียกเพื่อแสดงความสัมพันธ์ของคลาสที่อยู่ในลำดับชั้นดังนี้ คลาสที่เขียนไว้ด้านบนเรียกว่า *คลาสบน* (superclass) ส่วนคลาสที่เขียนไว้ด้านล่างเรียกว่า*คลาสล่าง* (subclass) เราสามารถพูดได้ว่า Account เป็นคลาสบนของ SavingAccount และ SavingAccount เป็นคลาสล่างของ Account

นอกจากใช้ศัพท์เรียกคลาสบน และคลาสล่างแล้ว ยังมีการใช้ศัพท์อื่นที่แตกต่างกันไปบ้างเช่น ใช้คำว่าคลาสแม่ (parent class) และคลาสลูก (child class) หรือคลาสฐาน (base class) และ คลาสอนุพันธ์ (derived class) ในความหมายเดียวกับคลาสบนและคลาสล่าง

คลาสล่างสามารถรับเอาตัวแปรและเมท็อดต่างๆ ที่กำหนดไว้ในคลาสบนมาใช้ในคลาสมันเองได้โดยไม่ต้องกำหนดซ้ำ การทำงานแบบนี้เองที่เรียกว่าการรับทอด คลาสล่างสามารถรับทอดเอา attribute (แทนด้วยตัวแปร) และพฤติกรรม (แทนด้วยเมท็อด) จากคลาสบนมาใช้ได้โดยไม่ต้องเขียนซ้ำในคลาสล่าง จึงเป็นการใช้ซ้ำ (reuse) สิ่งที่มีอยู่เดิมแล้วให้เกิดประโยชน์ยิ่งขึ้น ส่งผลให้มีผลิตภาพในการเขียนโปรแกรมสูงขึ้นนั่นเอง

ความสัมพันธ์แบบการรับทอดสามารถมีได้หลายลำดับชั้น คลาสล่างสามารถขึ้นไปรับทอดภาวะและพฤติกรรมจากคลาสด้านบนได้หลายชั้น ไล่ไปตามเส้นทางของมันที่เรียกว่า *สายรับทอด (inheritance chain)*



รูปที่ ๒ ผังลำดับชั้น แสดงความสัมพันธ์ของคลาสหลายระดับ

เช่นคลาส C1 รับทอดมาจากคลาส B1 และคลาส B1 รับทอดมาจากคลาส A เป็นตัวอย่างของ inheritance chain อีกตัวอย่างของ inheritance chain เช่น D2 รับทอดมาจาก C3 C3 รับทอดมาจาก B2 และ B2 รับทอดมาจาก A เป็นต้น คลาสล่างสามารถรับทอดภาวะและพฤติกรรมจากคลาสบนในลำดับชั้นต่าง ๆไล่ไปตาม inheritance chain จะรับทอดจากคลาสที่อยู่นอก inheritance chain ไม่ได้ เช่น D3 จะไปรับทอดจาก C4 ไม่ได้เพราะอยู่กันคนละ inheritance chain

จากรูปที่ ๒ คลาส A เป็นคลาสบนของทุกคลาสในรูปนี้ คลาส B1 เป็นคลาสบนของคลาส C1 และ C2 ส่งผลให้คลาส C1และ C2 เป็นคลาสล่างของ A ด้วย เราสามารถพูดได้ว่า คลาส C1 เป็นคลาสล่างของคลาส B1 และ A เมื่อเราพูดถึงคลาสบนจะหมายถึงคลาสที่อยู่เหนือขึ้นไปซึ่งอาจมีหลายคลาสในระดับที่แตกต่างลดหลั่นกันไป ในบางครั้งเราต้องการพูดเจาะจงถึงคลาสที่อยู่ในลำดับชั้นที่ติดกันเท่านั้น ในกรณีเช่นนี้เราจะเรียกคลาสที่อยู่เหนือขึ้นไปในลำดับชั้นที่ติดกันว่า*คลาสบนโดยตรง(direct superclass)* ส่วนคลาสล่างที่อยู่ในลำดับชั้นที่ติดกันลงไปเรียกว่า *คลาสล่างโดยตรง (direct subclass)* เช่น B1 เป็นคลาสบนโดยตรงของ C1 จะพูดในทิศทางกลับกันก็ได้ว่า C1 เป็นคลาสล่างโดยตรงของ B1

วิธีการเขียนคลาสล่าง

เราสามารถเขียนคลาสล่างได้เช่นเดียวกับคลาสทั่วไป แต่ต้องระบุเพิ่มอีกเล็กน้อยว่าคลาสล่างที่เขียนนั้นเป็นคลาสล่างของคลาสใด โดยเขียนชื่อคลาสบนไว้หลังคำสำคัญ `extends` ดังตัวอย่างเช่นถ้าต้องการให้ `B1` เป็นคลาสล่างของ `A` สามารถเขียนได้ตามรูปแบบดังนี้

```
class B1 extends A
{
    //   สมาชิกของคลาสละไว้ไม่ได้เขียน
}
```

หากต้องการเขียนคลาสล่างของ `B1` ลงไปอีกก็สามารถเขียนได้ทำนองเดียวกันดังนี้

```
class C1 extends B1
{
    //   สมาชิกของคลาสละไว้ไม่ได้เขียน
}
```

ซึ่งจะมีผลทำให้คลาส `C1` เป็นคลาสล่างของทั้ง `B1` และ `A`

เรามาดูตัวอย่างคลาสล่างที่ใช้งานได้จริง จากคลาส `Account` ที่เคยใช้เป็นตัวอย่างมาก่อนหน้านี้ ซึ่งเป็นคลาสบัญชีเงินฝากทั่วไป แต่ต่อไปนี้เราอยากแยกแยะบัญชีเงินฝากออกเป็นชนิดต่างๆ ให้ละเอียดยิ่งขึ้น เราอาจเขียนคลาสใหม่เช่น `SavingAccount` สำหรับใช้กับบัญชีออมทรัพย์เท่านั้น เราสามารถเขียนคลาสใหม่ให้เป็นคลาสล่างของ `Account` ได้ดังตัวอย่างที่ ๑

ตัวอย่างที่ ๑ แสดงการเขียนคลาส SavingAccount ให้เป็นคลาสล่างของ Account

```
๑. class Account {
๒.    int    number;
๓.    double balance;
๔.    void print() {
๕.      System.out.println("account number " + number +
๖.                    "; balance = " + balance);
๗.    }
๘.    void deposit (double amount) { balance += amount; }
๙.    void withdraw(double amount) { balance -= amount; }
๑๐. }
๑๑.
๑๒. class SavingAccount extends Account {
๑๓.
๑๔. }
๑๕.
๑๖. class Test {
๑๗.   public static void main( String args[]) {
๑๘.     SavingAccount sa = new SavingAccount( );
๑๙.     sa.number = 12345;
```

```
๒๐.     sa.print();
๒๑.     sa.deposit(5000);
๒๒.     sa.print();
๒๓.     sa.withdraw(3000);
๒๔.     sa.print();
๒๕.     sa.withdraw(4000.25);
๒๖.     sa.print();
๒๗.   }
๒๘. }
```

ตัวอย่างที่ ๑ แสดงการเขียนคลาสล่างอย่างง่าย โดยให้ SavingAccount เป็นคลาสล่างของ Account ตัวอย่างนี้ยังไม่ได้ใส่สมาชิกใด ๆ ให้ SavingAccount เพื่อสาธิตให้เห็นพื้นฐานของการรับทอด ถึงแม้เราจะไม่ได้กำหนดตัวแปรและเมท็อดใดลงในคลาส SavingAccount แต่ด้วยหลักการรับทอด คลาส SavingAccount นี้จะมีสมาชิกอยู่ในตัวมันด้วยจากการรับทอดมาจาก Account ซึ่งเป็นคลาสบนของมัน

SavingAccount จะรับทอดตัวแปร number และ balance มาไว้ในคลาสมันด้วย รวมทั้งเมท็อดต่าง ๆ ที่มีอยู่ในคลาส Account ก็เสมือนมีอยู่ในคลาส SavingAccount ด้วยเช่นเดียวกัน จะเห็นได้จากตัวอย่างนี้ที่มีการนำคลาส SavingAccount มาสร้างอ็อบเจกต์อ็อบเจ็กต์ในบรรทัดที่ ๑๘ แล้วนำค่า 12345 ไปใส่ไว้ในตัวแปร number ที่บรรทัด ๑๙ ถึงแม้ว่าในไม่ได้กำหนดให้มีตัวแปร number อยู่ในคลาส SavingAccount แต่สามารถใช้ตัวแปรนี้ได้เพราะ SavingAccount ได้รับทอดตัวแปร number จากคลาส Account มาใช้ในตัวมันด้วย บรรทัดที่ ๒๐ ก็ทำนองเดียวกัน ถึงแม้เราไม่ได้เขียนเมท็อด print ไว้ในคลาสนี้ก็ตาม เราสามารถเรียกใช้เมท็อด print ได้ราวกับว่าเราได้เขียนเมท็อดนี้ไว้ในคลาสนี้ด้วย การเรียกใช้เมท็อด deposite และ withdraw ก็เช่นเดียวกัน สามารถทำได้เนื่องจาก SavingAccount ได้รับทอดเอาเมท็อดเหล่านี้มาใช้ในตัวมันด้วยนั่นเอง

ผลลัพธ์ที่แสดงออกมาจากการทำงานของตัวอย่างที่ ๑ เป็นดังนี้

```
account number 12345; balance = 0.0
account number 12345; balance = 5000.0
account number 12345; balance = 2000.0
account number 12345; balance = -2000.25
```

การเขียนคลาสล่างโดยไม่ได้กำหนดให้มีตัวแปรหรือเมท็อดใดตามตัวอย่างนี้ดูไม่ค่อยมีประโยชน์ ซึ่งถ้าเป็นอย่างนี้เราสร้างอ็อบเจกต์จากคลาส Account โดยตรงมิดีกว่าหรือ วัตถุประสงค์ของตัวอย่างที่ ๑ ก็เพื่อต้องการแสดงให้เห็นว่า SavingAccount สามารถรับทอดเอาตัวแปรและเมท็อดที่กำหนดไว้ใน Account มาใช้ได้จริงๆ จึงไม่ได้ใส่ตัวแปรและเมท็อดใดๆ ลงไปให้ดูยุ่ง ในการเขียนคลาสล่างจริง เรามักจะใส่ตัวแปรสมาชิก และเมท็อดสมาชิกเพิ่มลงไปด้วย สมาชิกที่เขียนลงไปในคลาสล่างนี้ถือว่าเป็นการต่อเติม (extend) เพิ่มจากสมาชิกต่าง ๆ ที่มีอยู่แล้วในคลาสบนที่อยู่เหนือขึ้นไปทั้งหมด ซึ่งเท่ากับว่าเราขยายความสามารถของคลาสล่างให้มากกว่าคลาสบนนั่นเอง เพราะอย่างน้อยคลาสล่างมีความสามารถไม่ด้อยไปกว่าคลาสบนอยู่แล้วและยังมีการเพิ่มเติมสมาชิกอื่นเข้าไปอีก จึงพูดได้ว่าคลาสล่างมักมีความสามารถมากกว่าคลาสบน

ดูตัวอย่างที่ ๒ ซึ่งดัดแปลงมาจากตัวอย่างที่ ๑ โดยตัดเมท็อด `withdraw()` ออกจากคลาส `Account` เพราะเห็นว่าเมท็อด `withdraw()` ของ `Account` นำมาใช้กับ `SavingAccount` ไม่ได้ เนื่องจาก เมท็อด `withdraw()` ของคลาส `Account` นั้นไม่ได้ตรวจสอบว่ามีเงินคงเหลืออยู่ในบัญชีเพียงพอที่จะให้ถอนหรือไม่ ไม่ว่าจะถอนเท่าใด ก็ถอนได้ จะเห็นได้จากผลลัพธ์การทำงานของโปรแกรมตัวอย่างที่ ๑ ถอนหลายครั้งจนเงินคงเหลือในบัญชีติดลบ ซึ่งไม่ควรจะเป็นเช่นนั้น บัญชีสะสมทรัพย์ยอมให้ถอนได้จนหมดเงินที่มีอยู่ในบัญชี แต่จะถอนเกินกว่าเงินคงเหลือในบัญชีไม่ได้ เมื่อวิธีการถอนของบัญชีสะสมทรัพย์แตกต่างออกไปจากที่มีอยู่เดิมในคลาสบน เราจึงไม่อยากใช้วิธีการถอนของคลาส `Account` ในเมื่อวิธีการถอนของ `SavingAccount` เป็นวิธีการที่เฉพาะของคลาสนี้ เราสามารถเขียนเมท็อด `withdraw()` ขึ้นใช้เองในคลาส `SavingAccount` ก็ได้

ตัวอย่างที่ ๒ แสดงการต่อเติมเมท็อด `withdraw()` ลงไปในคลาส `SavingAccount`

```
๑. class Account {
๒.   int    number;
๓.   double balance;
๔.   void print() {
๕.     System.out.println("account number " + number +
๖.                  "; balance = " + balance);
๗.   }
๘.   void deposit (double amount) { balance += amount; }
๙. }
๑๐.
๑๑. class SavingAccount extends Account {
๑๒.  int withdraw(double amount) {
๑๓.    if (balance < amount)
๑๔.     return 1;  // return with error code 1
๑๕.    balance -= amount;
๑๖.  return 0;   // return with success
๑๗.  }
๑๘. }
๑๙.
๒๐. class Test {
๒๑.  public static void main( String args[]) {
๒๒.    SavingAccount sa = new SavingAccount( );
๒๓.    sa.number = 12345;
๒๔.    sa.print();
๒๕.    sa.deposit(5000);
๒๖.    sa.print();
๒๗.    sa.withdraw(3000);
๒๘.    sa.print();
๒๙.    sa.withdraw(4000.25);
๓๐.    sa.print();
๓๑.  }
```

```
๓๒. }
```

ตัวอย่างที่ ๒ ได้เขียนเมท็อด `withdraw()` เพิ่มเติมลงในคลาส `SavingAccount` ส่วนในคลาส `Account` ไม่มีเมท็อด `withdraw()` คราวนี้เป็นการใช้เมท็อดของคลาส `SavingAccount` เองไม่ได้รับทอดมาจาก `Account` แล้ว เมท็อด `withdraw()` ของคลาส `SavingAccount` นี้มีการตรวจสอบด้วยว่ามีเงินคงเหลืออยู่ในบัญชีเพียงพอให้ถอนหรือไม่ ถ้ามีไม่พอจะไม่ยอมให้ถอน จะกลับคืนไปสู่ผู้เรียกด้วยค่า 1 ชนิด `int` แต่ถ้าเงินคงเหลือในบัญชีมีเพียงพอคือไม่น้อยกว่า `amount` ก็จะยอมให้ถอนได้ เมื่อปรับลดยอด `balance` ลงแล้วจะกลับคืนไปพร้อมค่า 0 ค่าที่ส่งกลับคืนไปนี้จะเป็นประโยชน์กับผู้เรียกช่วยให้เมท็อดที่เรียกใช้เมท็อด `withdraw()` นี้ทราบว่าถอนได้สำเร็จหรือไม่ แต่ตัวอย่างนี้ไม่ได้นำค่าที่เมท็อด `withdraw()` คืนค่าให้ไปใช้ ทั้งนี้ เพื่อไม่ให้โปรแกรมยาวจนเกินไป แต่ผลจากการถอนไม่ได้จะทำให้เงินคงเหลือในบัญชีมีเท่าเดิม ไม่ให้ผลติดลบเหมือนอย่างตัวอย่างที่ ๑ ผลลัพธ์ที่ได้จากการทำงานของตัวอย่างที่ ๑ เป็นดังนี้

```
account number 12345; balance = 0.0
account number 12345; balance = 5000.0
account number 12345; balance = 2000.0
account number 12345; balance = 2000.0
```

การบดบัง (override) เมท็อด

ปกติแล้วคลาสล่างสามารถรับเอาเมท็อดในคลาสบนมาใช้ในคลาสล่างได้โดยอัตโนมัติ ซึ่งเป็นการใช้ประโยชน์จากเมท็อดของคลาสบนที่มีอยู่แล้ว แต่ในบางกรณี คลาสล่างไม่อยากใช้เมท็อดของคลาสบน ถ้าต้องการเช่นนี้เราต้องเขียนเมท็อดชื่อเดียวกันขึ้นมาใหม่ใส่ไว้ในคลาสล่าง ซึ่งเมท็อดในคลาสล่างจะถูกเรียกให้ทำงานแทนคลาสบน การทำเช่นนี้เท่ากับว่าคลาสล่างยกเลิกการใช้เมท็อดชื่อเดียวกันในคลาสบน การทำเช่นนี้ถือว่าเป็นการ override เมท็อด โดยที่เมท็อดในคลาสล่างได้ override เมท็อดที่มีอยู่เดิมในคลาสบน

การ override จะกระทำกับเมท็อดชื่อเดียวกันที่อยู่ต่างคลาสกันแต่ต้องอยู่ใน hierarchy chain เดียวกันและเมท็อดเมท็อดที่จะ override ต้องมีลายเซ็นต์พารามิเตอร์แบบเดียวกันด้วย ถ้ามีลายเซ็นต์พารามิเตอร์ต่างกันไม่ใช่การ override หรือถ้ามีลายเซ็นต์พารามิเตอร์เหมือนกันแต่อยู่ต่าง hierarchy chain ก็ไม่ถือว่าเป็นการ override เช่นกัน

ตัวอย่างโปรแกรมที่ ๓ แสดงการ override เมท็อด

```
๑. class Account {
๒.   int    number;
๓.   double balance;
๔.   void print() {
๕.     System.out.println("account number " + number +
๖.                  "; balance = " + balance);
๗.   }
๘.   void deposit (double amount) { balance += amount; }
๙. }
```

```
๑๐.
๑๑. class SavingAccount extends Account {
๑๒.   void print() {
๑๓.     System.out.println("Saving");
๑๔.   }
๑๕.   int withdraw(double amount) {
๑๖.     if (balance < amount)
๑๗.      return 1;  // return with error code 1
๑๘.     balance -= amount;
๑๙.   return 0;    // return with success
๒๐.   }
๒๑. }
๒๒.
๒๓. class Test {
๒๔.   public static void main( String args[]) {
๒๕.     SavingAccount sa = new SavingAccount( );
๒๖.     sa.number = 12345;
๒๗.     sa.print();
๒๘.     sa.deposit(5000);
๒๙.     sa.print();
๓๐.     sa.withdraw(3000);
๓๑.     sa.print();
๓๒.   }
๓๓. }
```

ดูตัวอย่างโปรแกรมที่ ๓ โปรแกรมนี้ปรับปรุงมาจากตัวอย่างโปรแกรมที่ ๒ โดยมีการเขียนเมท็อด `print()` เพิ่มลงในคลาส `SavingAccount` เมท็อด `print()` ตัวใหม่นี้ทำงานไม่เหมือน เมท็อด `print()` ของคลาส `Account` เมท็อดใหม่นี้จะแสดงข้อความว่า “Saving” เท่านั้น ไม่ได้แสดงเลขที่บัญชีและเงินคงเหลือแต่อย่างใด ในกรณีนี้เราไม่ต้องการใช้เมท็อด `print()` ของ `SavingAccount` ทำงานเหมือนเมท็อด `print()` ของ `Account` จึงทำการ override มันเสีย ผลลัพธ์จากการทำงานโปรแกรมนี้จึงเป็นดังนี้

```
Saving
Saving
Saving
```

บรรทัดที่ ๒๗ ๒๙ และ ๓๑ มีการเรียกใช้เมท็อด `print()` ผ่านตัวแปรอ้างอิง `sa` ซึ่งใช้อ้างถึงคลาส `SavingAccount` ในคลาส `SavingAccount` มีเมท็อด `print()` เป็นของมันเองอยู่แล้ว เมท็อด `print()` ของ `SavingAccount` จึงถูกใช้งาน ทำการพิมพ์ข้อความ “Saving” ออกมา เมท็อด `print()` ซึ่งอยู่ในคลาส `Account` ซึ่งอยู่ใน hierarchy chain เดียวกับคลาส `SavingAccount` และมีลายเซ็นต์พารามิเตอร์เหมือนกับของเมท็อด `print()` ในคลาส `SavingAccount` ดังนั้น เมท็อด `print()` ของคลาส `Account` จึงถูก override ด้วยเมท็อด `print()` ของคลาส `SavingAccount` ซึ่งส่งผลให้เมท็อด `print()` ของคลาส `Account` ไม่ถูกเรียกใช้

ตัวอ้างอิง super

ในบางกรณีคลาสล่างอาจต้องการเรียกใช้เมท็อดของคลาสบน เราสามารถเรียกใช้เมท็อดในคลาสบนได้โดยเรียกใช้ผ่านทางตัวอ้างอิง `super` ดังตัวอย่างที่ ๔

ตัวอย่างที่ ๔ แสดงการใช้ตัวอ้างอิง super ในการเรียกใช้เมท็อดของคลาสบน

```
๑. class Account {
๒.   int    number;
๓.   double balance;
๔.   void print() {
๕.     System.out.println("account number " + number +
๖.                    "; balance = " + balance);
๗.   }
๘.   void deposit (double amount) { balance += amount; }
๙. }
๑๐.
๑๑.class SavingAccount extends Account {
๑๒.  void print() {
๑๓.    System.out.print("Saving ");
๑๔.    super.print();
๑๕.  }
๑๖.  int withdraw(double amount) {
๑๗.    if (balance < amount)
๑๘.     return 1;  // return with error code 1
๑๙.    balance -= amount;
๒๐.  return 0;   // return with success
๒๑.  }
๒๒.}
๒๓.
๒๔.class Test {
๒๕.  public static void main( String args[]) {
๒๖.    SavingAccount sa = new SavingAccount( );
๒๗.    sa.number = 12345;
๒๘.    sa.print();
๒๙.    sa.deposit(5000);
๓๐.    sa.print();
๓๑.    sa.withdraw(3000);
๓๒.    sa.print();
๓๓.  }
๓๔.}
```

จากตัวอย่างที่ ๓ เมท็อด `print()` ของ `SavingAccount` พิมพ์เพียงข้อความว่า “Saving” เท่านั้น เราอยากให้เมท็อดนี้แสดงเลขที่บัญชีและจำนวนเงินคงเหลือออกมาด้วย แต่เนื่องจากมีเมท็อด `print()` ในคลาส

Account ทำหน้าที่นี้อยู่แล้ว เราไม่จำเป็นต้องเขียนเมท็อด print() ในคลาส SavingAccount ให้ทำงานเหมือนกับเมท็อด print() ของ Account ให้ซ้ำซ้อน เมท็อด print() ของ SavingAccount สามารถเรียกใช้ เมท็อด print() ของ Account ให้ทำงานแทนได้

ตัวอย่างที่ ๔ ดัดแปลงมาจากตัวอย่างที่ ๓ เพียงเล็กน้อย โดยแก้เมท็อด print() ของคลาส tSavingAccount. ให้เรียกใช้เมท็อด print() ของคลาส Account บรรทัดที่ ๑๔ ข้อความสั่ง super.print() เป็นการเรียกใช้เมท็อด print() ของคลาสบน ซึ่งในที่นี้คือคลาส Account มีการปรับปรุงบรรทัดที่ ๑๓ เล็กน้อยโดยเปลี่ยนจาก การเรียกเมท็อด println() เป็น print() เพื่อไม่ให้มีการขึ้นบรรทัดใหม่หลังจากพิมพ์แล้ว เพราะต้องการให้แสดงเลขที่บัญชีและเงินคงเหลือของบัญชีจากเมท็อด print() ของ Account ต่อเนื่องในบรรทัดเดียวกัน ผลลัพธ์จึงออกมาดังนี้

```
Saving account number 12345; balance = 0.0
Saving account number 12345; balance = 5000.0
Saving account number 12345; balance = 2000.0
```

ตัวอย่างนี้อาจไม่เห็นประโยชน์ของการใช้ super มากนัก เนื่องจากเป็นตัวอย่างโปรแกรมที่ค่อนข้างสั้นการเรียกใช้เมท็อดของคลาสบนจะมีประโยชน์มากถ้าเมท็อดของคลาสบนที่ต้องการเรียกนั้นมีความยาวมาก การใช้ super ช่วยในการเรียกเมท็อดของคลาสบนนี้ช่วยให้เราหลีกเลี่ยงการเขียนเมท็อดที่ทำงานซ้ำซ้อนกัน ซึ่งนอกจากจะเสียเวลาในการเขียนแล้ว ยังสร้างความยุ่งยากในการแก้โปรแกรมหลายที่ในภายหลังด้วย

ในกรณีคลาสล่างต้องการเรียกใช้ตัวสร้างของคลาสบน สามารถทำได้โดยใช้ตัวอ้างอิง super ช่วยดังตัวอย่างที่ ๕

ตัวอย่างโปรแกรมที่ ๕ แสดงการใช้ super เรียกตัวสร้างของคลาสบน

```
๑. class Account {
๒.    private int    number;
๓.    double balance;
๔.    Account(int number) { this.number = number; }
๕.    void print() {
๖.      System.out.println("account number " + number +
๗.                    "; balance = " + balance);
๘.    }
๙.    void deposit (double amount) { balance += amount; }
๑๐. }
๑๑.
๑๒. class SavingAccount extends Account {
๑๓.   SavingAccount(int number) { super(number);  }
๑๔.   void print() {
๑๕.     System.out.print("Saving ");
๑๖.   super.print();
๑๗.   }
๑๘.   int withdraw(double amount) {
```

```
๑๙.     if (balance < amount)
๒๐.      return 1;  // return with error code 1
๒๑.     balance -= amount;
๒๒.  return 0;   // return with success
๒๓.   }
๒๔. }
๒๕.
๒๖. class Test {
๒๗.   public static void main( String args[]) {
๒๘.     SavingAccount sa = new SavingAccount( 12345);
๒๙.     // sa.number = 12345;
๓๐.     sa.print();
๓๑.     sa.deposit(5000);
๓๒.     sa.print();
๓๓.     sa.withdraw(3000);
๓๔.     sa.print();
๓๕.   }
๓๖. }
```

ตัวอย่างที่ ๕ ดัดแปลงมาจากตัวอย่างที่ ๔ ให้เพิ่มการปกป้องข้อมูลมากขึ้นและมีการใช้ตัวสร้างในการกำหนดค่าเริ่มต้นให้กับตัวแปรสมาชิก ที่บรรทัดที่ ๒ เพิ่มตัวดัดแปร private ให้ตัวแปร number จะส่งผลให้ตัวแปรนี้สามารถเข้าถึงได้เฉพาะภายในคลาส Account เท่านั้น ดังนั้น บรรทัดที่ ๒๙ ที่มีการเข้าถึงตัวแปร number จากคลาส Test จึงทำไม่ได้อีกต่อไป จึงได้คอมเม้นต์ไว้ เราจะใช้วิธีส่งเลขบัญชีให้ตัวสร้างไปดำเนินการแทน ที่บรรทัดที่ ๒๘ มีการส่งเลขที่บัญชีให้ตัวสร้าง โดยเจตนาให้ตัวสร้างเป็นผู้นำเลขบัญชีนี้ไปใส่ในตัวแปร number

ตัวอย่างนี้จึงมีการเขียนตัวสร้างให้กับคลาส SavingAccount ดังแสดงในบรรทัดที่ ๑๓ ตัวสร้างตัวนี้รับพารามิเตอร์ ๑ ตัวคือเลขที่บัญชี ซึ่งมันควรจะนำเลขบัญชีที่รับมานี้ไปใส่ในตัวแปร number แต่เนื่องจากตัวแปร number ประกาศไว้ในคลาส Account ให้เป็น private ตัวสร้าง SavingAccount() จึงไม่สามารถเข้าถึงตัวแปร number ได้ (ถ้าประกาศตัวแปร number ให้เป็น protected ตัวสร้าง SavingAccount() จึงจะเข้าตัวแปรสมาชิก number ถึงได้) เราจึงใช้วิธีส่งเลขที่บัญชีไปให้ตัวสร้างของคลาส Account ดำเนินการแทน

นิพจน์ super(number) เป็นการเรียกใช้ตัวสร้างของคลาสบนของ SavingAccount ซึ่งก็คือ ตัวสร้าง Account นั่นเอง โดยส่งเลขที่บัญชีซึ่งเพิ่งรับเข้ามาไปให้ด้วย ส่วนตัวสร้าง Account() เขียนไว้ที่บรรทัดที่ ๔ ตัวสร้างตัวนี้รับเลขที่บัญชีเข้ามาในรูปของพารามิเตอร์ number แล้วนำค่าที่รับมานี้ไปใส่ไว้ในตัวแปรสมาชิก number ของอ็อบเจ็กต์ปัจจุบัน ซึ่งหมายถึงตัวแปร number ของอ็อบเจ็กต์ซึ่งอ้างถึงโดยตัวแปรอ้างอิง sa ซึ่งมีชนิดเป็น SavingAccount ถึงแม้ในคลาส Account จะประกาศตัวแปร number ให้เป็น private คลาส SavingAccount ก็ยังรับทอดมาไว้ในอ็อบเจกต์ของคลาสมันได้ แต่ว่าจะเข้าถึงเพื่ออ่านหรือเปลี่ยนแปลงค่าของ number เองโดยตรงไม่ได้ ถ้าต้องการเข้าถึงต้องกระทำผ่านเมท็อดของคลาส Account เท่านั้น