หลักการออกแบบและใช้งานคลาส

Actor

- ให้เราสร้างออบเจ็กต์ได้
- แล้วทำงานอย่างใดอย่างหนึ่งให้กับโปรแกรมอื่นๆได้
- เช่น Scanner, Random

Utility

- มีแต่ค่าคงที่กับstatic method ให้เรียกใช้
- เช่น Math

หลักการออกแบบ

- ชื่อคลาสต้องเป็นคำนาม
- ชื่อคลาสต้องสื่อให้เรารู้ว่าจะทำอะไรกับออบเจ็กต์ของคลาสนั้ นได้บ้าง
  - ตัวอย่าง ถ้าทำโปรแกรมเกี่ยวกับการจ่ายเงินเดือน เราไม่ควรสร้างคลาสคลาสเดียวที่ทำทุกอย่าง อย่าง PaycheckProgram
  - แต่เราควรเขียน Paycheck ขึ้ นมา แล้วนิยามเมธอดที่เปลี่ยนค่าหรือจัดการ Paycheck ได้ แล้วจึงนำ Paycheck ออบเจ็กต์ไปใช้ในคลาสที่เรารัน main

- ถ้าเกิดคลาสมีเมธอดที่ทำหน้าที่ไม่ค่อยเกี่ยวข้องกัน เราควรแยกคลาสนั้นออกเป็นสองคลาส ให้แต่ละคลาสเรียกเมธอดที่เกี่ยวกันภายในคลาสเท่านั้น
  - ทำให้โปรแกรมดูง่ายขึ้น แก้ง่ายขึ้น
  - แต่ว่าคลาสหนึ่งจะต้องใช้อีกคลาสหนึ่ง ถ้ามีมากๆอาจเป็นปัญหา
  - ดังนั้นให้แยกคลาสเท่าที่จำเป็นเท่านั้น

- เวลาเขียนเมธอดต่างๆ ถ้าเป็นเมธอดที่มีหลักการทำงานเหมือนๆกันอยู่แล้ว หัวเมธอดก็ต้องเขียนให้เป็นแนวเดียวกัน

Immutable class

- เป็นคลาสที่มีแต่ accessor เมธอดเท่านั้ น
  - ตัวอย่างคือ String เมื่อออบเจ็กต์ที่เป็น String ถูกสร้างขึ้ นมาแล้ว จะไม่มีวัน เปลี่ยน เพราะไม่มีเมธอดในคลาส String ที่จะเปลี่ยนค่าอะไรของString ออบเจ็กต์ได้เลย

```
String name = “John”;
String uppercase = name.toUpperCase(); // name ไม่เปลี่ยนนะ
```

- ข้อดีของคลาสประเภทนี้ คือ เอาref ของออบเจ็กต์ให้คนอื่นใช้งานได้โดยไม่ ต้องกังวลว่าค่าจะเปลี่ยน

Side effect

- เมธอดที่มี side effect คือ เมธอดที่โค้ดของมันเปลี่ยนค่าที่มองเห็นได้ แม้ว่าเมธอดจะรันเสร็จไปแล้ว เช่น ค่าของ instance variable

ตัวอย่างเช่น

```
public class BankAccount{
….
   public void transfer(double amount, BankAccount other){
         balance = balance –amount;
         other.deposit(amount);
   }
}
```

เปลี่ยนค่าตัวแปรใน this

เปลี่ยนค่าตัวแปรในออบเจ็กต์อีกตัวด้วย

ข้อควรระวัง

- ถ้าใช้ออบเจ็กต์เป็นพารามิเตอร์ของเมธอด เราเปลี่ยนค่าภายในออบเจ็กต์นั้ นได้ การเปลี่ยนแปลงจะยังอยู่แม้ว่าเมธอดจะรันเสร็จแล้ว
- แต่ว่า ถ้าพารามิเตอร์ของเมธอดเป็น primitive type เราจะเปลี่ยนค่ายังไง ก็จะมีผลอยู่แค่ในเมธอดเท่านั้ น
- ดูตัวอย่าง

- สมมุติเราเปลี่ยนเมธอด transfer เป็นแบบนี้

```
public void transfer(double amount, double otherBalance){
        balance = balance –amount;
        otherBalance = otherBalance + amount;
   }
```

ตอนเข้าไปรันเมธอด มันจะถือว่ารับค่าเข้าไปเท่านั้ น ตัวต้นฉบับไม่เปลี่ยน

แล้วไปเรียกใช้งานแบบนี้

```
double savingBalance = 1000;
myAccount.transfer(500,savingBalance);
System.out.println(savingBalance);
```

อย่าใช้พารามิเตอร์เป็นที่ทด

- ถึงเราจะรู้ว่า พารามิเตอร์ในเมธอด ที่เป็นprimitive type นั้ น จะไม่มีผลกับภายนอก แต่ก็ไม่ควรใช้มันเป็นที่ทด

```
public void deposit(double amount){
    amount = balance + amount;
    …
}
```

อย่างนี้ จะทำให้พารามิเตอร์ที่เราเอาเข้ามา เปลี่ยนค่า ทำให้ใช้ค่าผิดจากที่ต้องการได้ภายในเมธอดนี้ ทำให้เพื่อนที่มาใช้โปรแกรมเราต่อสับสนด้วย

ถ้าต้องการทด สร้าง local variable ขึ้ นมาใหม่ภายในเมธอดเลยดีกว่า

ข้อควรระวังอีกอย่าง

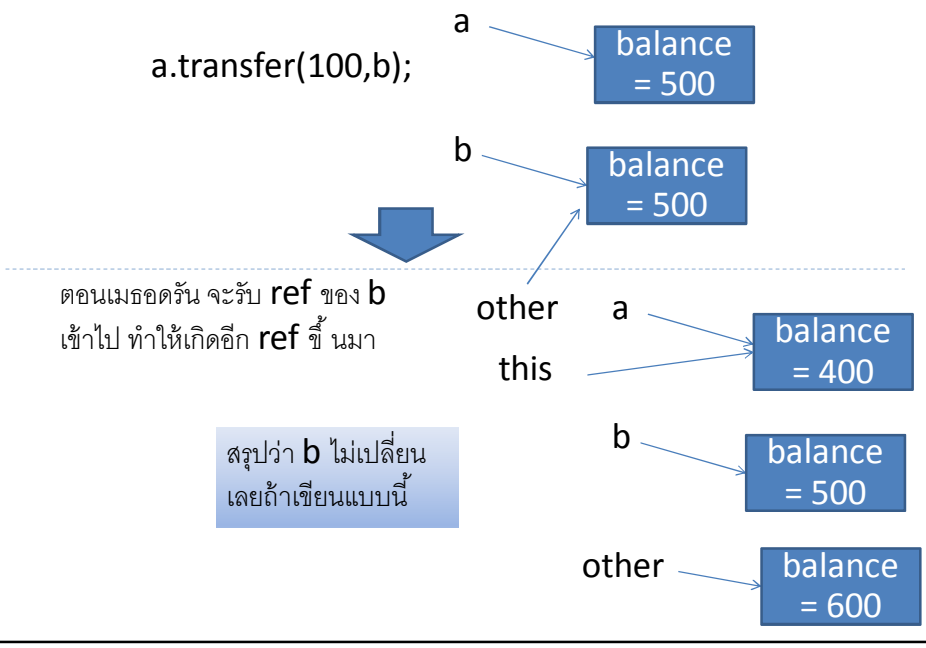
- ระวังจะลบหรือเปลี่ยนข้อมูลที่ไม่สมควรเปลี่ยน เช่น

```
public class GradeBook{
    public void addStudents(ArrayList<String> names){
        while(names.size()>0){
            String a_name = names.remove(0);
            add(a_name);
        }
    }
}
```

ตัวอาร์เรย์ลิสต์ต้นฉบับจะถูกลบด้วย ดังนั้ นจะเอาไปใช้งานที่อื่นในโปรแกรมอีกไม่ได้แล้ว

ข้างล่างนี้ผิด เพราะอะไร?

```
public class BankAccount{
….
   public void transfer(double amount, BankAccount
   other){
        balance = balance –amount;
        double newBalance = other.balance +amount;
        other = new BankAccount(newBalance);
   }
}
```



precondition

- คือสิ่งที่ต้องเป็นจริงก่อนที่เมธอดจะถูกเรียกใช้ ถ้าไม่เป็นจริง เจ๊งแน่
- เช่น เมธอด deposit ต้องรับเลขที่ไม่เป็นลบเท่านั้ น
- ตามปกติ ควรจะเขียนเป็นส่วนหนึ่งของคอมเม้นไว้ จะได้เอาไว้เช็ค

```
/**
        Deposit money tothe bank account.
        @param amount the amount to deposit
        (Precondition: amount >=0)
*/
public void deposit(double amount){…}
```

- ถ้าเมธอดถูกเรียกโดยที่ precondition ของมันยังไม่เป็นจริง
  - เราอาจตรวจแล้ว throw exception ได้ (เดี๋ยวได้เรียนเอง)
  - หรือเราอาจปล่อยเลยตามเลย เพราะถือว่าคนเรียกเมธอดผิดเองที่ไม่ได้เรียกตาม precondition
  - เดี๋ยวก่อน ยังมีอีกวิธีสำหรับ java

assertion

```
Public double deposit(double amount){
   assert amount >= 0;
   balance = balance + amount;
}
```

สามารถสั่ง enable/disable ได้ ทำให้แยกส่วนนี้ ออกจากโปรแกรมได้

ถ้า enable อยู่ แล้วได้ false โปรแกรมจะหยุดทำงาน แล้ว throw AssertionError

Java –ea MainClass

- ทำให้ error ชัดเจนแบบนี้ จะได้รู้ตัวว่าเขียนผิด และรีบแก้
- ดีกว่ามา if แล้ว return กลับ โดยไม่ทำอะไร เพราะอันนี้ คนรันจะไม่เห็นว่าโปรแกรมผิดเลย

postcondition

- ได้ค่า return value ที่ถูกต้อง
- ออบเจ็กต์เปลี่ยนไปตามที่ต้องการ

```
/**
        Deposit money tothe bank account.
        (Postcondition: getBalance() >= 0)
        @param amount the amount to deposit
        (Precondition: amount >=0)
*/
public void deposit(double amount){…}
```

Class invariant

- คือสิ่งที่เป็นจริงเสมอหลังจากสร้างออบเจ็กต์แล้ว ไม่ว่าเมธอดอะไรจะโดนเรียกก็เป็นจริงอยู่
- ถ้าเรารู้ว่าclass invariant คืออะไร เราก็ตรวจโปรแกรมเราได้ว่าขณะนั้ นเขียนถูกอยู่หรือไม่

มาดูโค้ดของ BankAccount กันอีกที

```
public class BankAccount{

double balance;

/**
    Construct a bank account with a given balance.
    @param initialBalance the money in the account at the time it is open.
    (Precondition: initialBalance >=0)
*/
public BankAccount(double initialBalance){
    balance = initialBalance;
}

/**
    Deposit money tothe bank account.
    (Postcondition: getBalance() >= 0)
    @param amount the amount to deposit
    (Precondition: amount >=0)
*/
public void deposit(double amount){…}
```

```
/**
    Withdraws money from the bank account.
    @param amount the amount to withdraw
*/
public void withdraw(double amount){…}


/**
    Gets the current balance of the bank account.
    @return the current balance
*/
public double getBalance(){…}
```

- เราสังเกตได้ว่า ตั้ งแต่สร้างออบเจ็กต์ ค่าgetBalance() >= 0 เสมอแน่นอน ดูจากprecondition ของทุกเมธอด แล้วก็สังเกตดูด้วยว่าหลังจากแต่และเมธอดถูกเรียกแล้ว ยังเป็นจริงอยู่หรือเปล่า
- เอาค่า invariant นี้ ไปเขียนบรรยายเป็นคอมเม้นซะ

```
/**
   …
   (Invariant: getBalance() >= 0)
*/
public class BankAccount{…}
```

Static method

- ไม่ต้องสร้างออบเจ็กต์ ก็เรียกใช้ได้เลย เช่นเมธอดใน utility class อย่าง Math
- สร้างขึ้ นเพื่อลดการเขียนโค้ด อย่างที่เรียนตอนปีหนึ่งนั่นแหละ แต่เมธอดแบบนี้ จะไม่ต้องใช้เปลี่ยนหรืออ่านค่าของออบเจ็กต์ไง
- หลายๆเมธอดเป็น static เพราะว่ามันเล่นกับprimitive type เช่นตัวเลข ซึ่งเราสร้างออบเจ็กต์ไม่ได้
- เรียกเมธอดประเภทนี้ ต้องเรียกจากการใช้ชื่อคลาส

การใช้งานอีกอย่างของ static method

- ใช้เมื่อเราไม่ต้องการเปลี่ยนคลาสที่มีอยู่แล้ว หรือเปลี่ยนคลาสที่มีอยู่แล้วไม่ได้

```
public class Geometry{
   public static double area(Rectangle rect){
        return rect.getWidth()*rect.getHeight();
   }
}
```

เป็นของจาวา เราเปลี่ยนไม่ได้ เลยรับเป็นพารามิเตอร์ของเมธอดของคลาสใหม่แทน ให้เมธอดเป็น static เพราะเรามีออบเจ็กต์อยู่แล้ว คือตัว Rectangle

หรือใช้เพื่อ access/mutate ตัวแปรที่เป็น static

- ใช้สำหรับจัดการกับตัวแปรของคลาส ที่แชร์กันระหว่างทุกออบเจ็กต์ในคลาสนั้น

วิธี initialize static variable

- ไม่ทำอะไร มันก็มีค่าdefault เอง
- Assign ค่า ตอนที่เรา declare มัน

```
private static int x = 1000;
```

- อีกวิธีคือใช้ static initialization block

Static import

```
import static java.lang.System.*;
import static java.lang.Math.*;

double r = sqrt(PI);
out.println(r);
```

ทำให้ไม่ต้องเขียนยาว
และโค้ดอ่านง่ายขึ้ น

วิธี initialize instance variable

- ใช้ค่า default
- ใช้ constructor
- เขียนค่า default เอง ตอน declare ตัวแปรนั้ น

```
public class Coin{
   private double value =1;
   …
}
```

ค่านี้ จะเป็นdefault ตอนสร้างทุกออบเจ็กต์ของคลาสนี้

- ใช้ initialization block

```
public class Coin{
   private double value ;
   static int shared;
   {
        value =1;
   }

   static
   {
        shared = 1;
   }

   …
}
```