Software Construction
Final Lecture

Edit 6.0

17/12/2016

By Aely

Contents

คำเตือน

สรุปฉบับนี้มีเนื้อหาวกวน เวิ่นเว้อ เพ้อเจ้อ และไร้สาระเป็นจำนวนมาก โปรดทำใจก่อนอ่าน และอย่าด่ากู กูแค่ว่าง ขอบคุณครับ

Part I : Polymorphism

เป็นคุณสมบัติหนึ่งของ OOP คือ อะไรสักอย่าง พิมพ์ไม่ถูก ไปดูตัวอย่างแล้วมโนเอาเองล่ะกัน เหตุการณ์ต่างๆที่ใช้คุณสมบัติของ Polymorphism มีดังนี้

กำหนด class B extends A

Scenario 1 : การใช้เป็น static type

Ex. A a = new B();

เรื่องคุณสมบัติว่าทำแบบนี้แล้วจะเป็นยังไง ไปอ่านอันเก่าๆเอาเนอะ

Note โดยปกติชื่อ class ข้างหน้ากับข้างหลังจะเป็นตัวเดียวกัน กรณีเดียวที่เป็นคนละคลาสได้คือ คลาสด้านหน้าเป็น superclass ของด้านหลัง

Scenario 2 : ใช้เป็น parameter type

Ex. public void method(A a){ }

method(new B());

- เมธอดที่ประกาศรับ superclass สามารถใส่ obj. ของ subclass ไปแทน

Note คุณสมบัติแบบนี้จะใช้บ่อยในเรื่อง Interface

Scenario 3 : ใช้เป็น type ของ Array / ArrayList

Ex. ArrayList<A> list1 = new ArrayList<>();

list1.add(new B());

Ex. A[] list2 = new A[2];

list2[0] = new B();

Sarup

ตรงตำแหน่งที่มีการประกาศ type ของตัวแปร เป็น superclass แต่ค่าที่ใส่ให้สามารถเป็น subclass ได้

Part II : Interface

คุณสมบัติ

- ทุก method จะเป็น *public abstract* โดยอัตโนมัติ
- ไม่มี instance variable
- attribute ที่ประกาศไว้มีสมบัติเป็น *public static final* โดยอัตโนมัติ
- ไม่สามารถสร้าง obj. ได้

Example Code

```
public interface Drawable{
        void draw(Graphics2D g2);
}
public class Car implements Drawable{
        public void draw(Graphics2D g2){
                ...
        }
}
```

Algorithm reuse

สมมติโค้ดหน้าตาแบบนี้

```
public static sum(Account[] accs){
        double sum = 0;
        for(Account acc : accs))
                sum += acc.getBalance();
        return sum;
```

```
Public static sum(Town[] towns){
        double sum = 0;
        for(Town town : towns)
                sum += town.getArea();
        return sum;
```

เบื่อมั๊ยกับการเขียนโค้ดเหมือนๆกันซ้ำๆซากๆ เบื่อมั๊ยกับการก๊อปแล้ววาง นี้ขนาดมีแค่ 2 คลาส ต้องเขียนเมธอดเพื่อทำงานกับแต่ล่ะคลาสแบบนี้ตั้ง 2 อัน ถ้ามีเป็นสิบๆคลาส ตายห่าพอดี(ตอนทำไม่ตายหรอก แค่ก๊อปวางนิ คิดถึงตอนแก้สิ หาวนไปจ้ะ)

วันนี้เราขอเสนอ **Interface** ของวิเศษที่ช่วยให้เขียนโค้ดง่ายขึ้น(มั้ง)

แก้โค้ดใหม่

step 1 สร้าง interface

```
public interface Measurable {
        double getMeasure();
}
```

step 2 Implements interface

```
public class Account implements Mea..{
        public double getMeasure(){
                return getBalance();
        }
}
```

```
public class Town implement Measurable {
        public double getMeasure(){
                return area;
        }
}
```

Step 3 กลับไปแก้ sum

```
public static sum(Measurable[] meas){
        double sum = 0;
        for(Measurable mea : meas))
                sum += mea.getMeasure();
        return sum;
```

ว้าว เห็นมั๊ยล่ะครับ แค่นี้เมธอด sum แค่เมธอดเดียวก้เพียงพอแล้ว

Converting

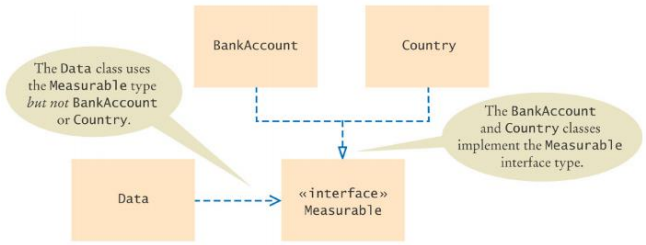
```
กำหนด  interface Hero {        void fly();     }
       class Person {           void walk();    }
       class Ironman extends Person implements Hero { }
```

```
IronMan im1 = new Ironman();        // OK
Person im2 = new Ironman();         // OK
Hero im3 = new Ironman();           // OK
Hero im4 = im1;                     // OK
Hero im5 = (Hero) im2;              // OK
Hero im6 = new Person();            // ERROR    Person ไม่ได้เกี่ยวไรกับ Hero เลย
```

```
Ironman im7 = (Ironman) im4;        // OK       im4 เก็บไว้เป็น Hero อย่าลืม cast
Hero im8 = new Hero();              // ERROR    interface สร้าง obj. ไม่ได้

im1.fly();      // OK
im1.walk();     // OK
im2.fly();      // ERROR    im2 มี static type เป็น Person และ Person ไม่มี fly
im3.walk();     // ERROR    im3 มี static type เป็น Hero และ Hero ไม่มี walk
```

Diagram



- สังเกต เส้นของ interface จะเป็นเส้นประ และหัวจะเป็นหัวขาว

นิทาน Java มหาสนุก ตอน Interface

ถ้าไม่ใช้ interface Data จะต้องโยงไปหาทุกๆคลาส ซึ่งมันแย่มาก ตัวอย่างให้เห็นภาพแบบง่ายๆ สมมติเขียนโค้ดกัน 2 คน นาย A เขียนคลาส Data นาง B เขียนที่เหลือ

แบบไม่ใช้ interface

นาย A เขียนโค้ดเสร็จเรียบร้อย กำลังจะไปเก็บกระเป๋าไปเที่ยวฮาวาย แน่นอนใน method sum(Account[] accs) นาย A ใช้ acc.getBalance() แบบโค้ดในตอนแรก

อยู่ดีๆ นาง B ก็เกิดกวนตีนอยากเปลี่ยนชื่อ method getBalance เป็น getCurrent

นาย A ที่กำลังจะไปฮาวายก็เลยต้องมาเปิดคอมแล้วเปลี่ยนโค้ดให้เป็น acc.getCurrent()

แบบใช้ interface

นาย A เขียนโค้ดเสร็จตามเดิม กำลังจะเก็บกระเป๋าไปเที่ยวภูเก็ต คราวนี้ใน method sum(Measure[] meas) นาย A ใช้แค่ mea.getMeasure() แบบโค้ดอันหลัง

อยู่ดีๆ นาง B ก็เกิดกวนตีนอีกครั้ง อยากเปลี่ยนชื่อ method เป็น getCurrent อีกแล้ว แต่คราวนี้นาง B ก็แค่ไปแก้ใน getMeasure ด้วย ส่วนนาย A ก็ไปเที่ยวภูเก็ตได้ เพราะโค้ดฝั่งของ Data ยังเหมือนเดิม

สรุป การใช้ interface ช่วยให้ทำงานกับชาวบ้านง่ายขึ้น แม้จะกวนตีนก็ตาม จบจ้า

Part III : Interface Call Back

Structure

นิทาน Java มหาสนุก ตอน Interface Call Back 1

จากนาย A กับ นาง B คู่เดิม คราวนี้นาง B มาพร้อมกับ class Town ที่มี Attribute area population และ gdp

คราวนี้ปัญหามาจากนาย A เสือกอยากสร้างเมธอดที่ sum ได้หลายค่า แต่ไอ่ getMeasure แบบข้างบนมันคืนค่าได้แค่ค่าเดียวเนี่ยสิ

Problem คลาสบางคลาสมีค่าที่ต้องการใช้มากกว่า 1 ค่า

Solution ถ้าถาม Town ตรงๆล่ะมันเลือกไม่ถูกว่าจะตอบอะไร งั้นก้สร้างคนถามขึ้นมาหลายๆคนสิ

step 1 สร้าง interface Measurer

```
interface Measurer
        double measure(Object o);
```

step 2 implement Measurer ต่างๆ สำหรับถามค่าต่างๆ

```
class AreaMeasurer implements Measurer
        public double measure(Object o)
                return ((Town) o).getArea();
class PopulationMeasurer implements Measurer
        public double measure(Object o)
                return ((Town) o).getPopulation();
class GDPMeasurer implements Measurer
        public double measure(Object o)
                return ((Town) o).getGdp();
```

step 3 แก้เมธอด sum ใหม่

```
public static double sum(Object[] os, Measurer mea){
        double sum = 0;
        for(Object o : os)
                sum += mea.measure(o);
```

นิทาน Java มหาสนุก ตอน Interface Call Back 2

นาย A กับ นาง B คู่เดิม อยากได้คลาส Paper ที่มี attribute size เลยไปฝากให้ นาย C ช่วยทำให้ หลังจากนาย C ทำให้เสดก็บินหนีไปฮันนี่มูนที่ญี่ปุ่น ปัญหาเกิดตอนที่พอนาย A จะเอา Paper มาใช้ แต่นาย C ดันลืม implements Measurable (กำหนดให้ไม่มีใครสามารถแก้โค้ดของชาวบ้านได้)

Problem คลาสบางคลาสเราก็ไม่ได้เขียนเอง จะให้ไปแก้โค้ด implement interface ก็ไม่ได้ไง

Solution

step 1 สร้าง Measurer ของแต่ล่ะค่าขึ้นมา

```
class SizeMeasurer implements Measurer
        public double measure(Object o)
                return ((Paper) o).getSize();
```

step 2 อ้าวเสร็จแล้วนี้หว่า หมดล่ะ

นิทาน Java มหาสนุก ตอน Interface Call Back 3

นาย A กำลังเขียนโปรเจค nimble quest อยู่ ตอนนี้กำลังคิดว่าจะให้ Hero มี เมธอด attack เพื่อใช้โจมตี Monster แถมเวลาโจมตีเนี่ย ยังสามารถใช้อาวุธได้หลายแบบด้วย แล้ววิธีโจมตีก็แตกต่างไปตามอาวุธที่ใช้ ว้าว โปรเจคเว่อวังมาก

นาย A จึงลองเอาโปรเจคไปเสนอ ปรมจารย์ Rujji ปรมจารย์สนใจเป็นอย่างมาก เลยบอกว่าอยากได้อาวุธสัก 88 แบบ ถ้าทำได้ จะมอบ 80 คะแนนให้ นาย A ตกใจมากจึงไปขอให้แม่นาง B ช่วยตีอาวุธให้สัก 88 แบบ

เด๋วมาต่อ

Part IV : Anonymous Class & Inner Class

Anonymous Syntax

Interface	Abstract class

new InterfaceName(){
@Override
…
};
	New AbstractName(arg0, arg1, …){
	@Override
	…
	};

Inner Class Scope

สร้างใน method ใช้สร้าง obj. และอ้างอิง(ประกาศ type) ได้เฉพาะใน method นั้น

```
class A {
    public void methodA(){
        class B { … }
        B b1 = new B();
    }
    public void methodB(){
        B b2 = new B();
    }
```

สร้างใน class แล้วแต่ access modifier

class ที่มี inner class เรียกว่า outer class

Part V : Generic Type

นิทาน Java มหาสนุก ตอน Generic Type

นาง B เรียนเรื่อง Graph มาจากวิชา Data Structure นาง B จึงอยากลองเอามาเขียน Java ทีนี้ที่เรียนมามันต้องมี class Node และมี attribute data เพื่อเก็บข้อมูล และมีลิสต์เพื่อในเก็บ Node ที่สามารถไปหาได้ แต่นาง B ต้องการสร้าง Node ที่ไว้เก็บข้อมูลประเภทอะไรก็ได้ แต่ว่าตอนที่จะสร้าง data ต้องประกาศ type ของมันนี้สิ

นาย A ที่เพิ่งกลับมาจากภูเก็ตเห็นนาง B กำลังนั่งเครียด เลยเข้าไปถามว่าเป็นอะไร จากนั้นจึงได้หยิบของวิเศษออกมาจากกระเป๋ามิติที่ 4 นั่นก็คือ... **Generic Type**

Problem อยากสร้าง class ที่ยังไม่รู้ชนิดของข้อมูลภายใน class

Solution

```
class Node<T>{
        private T data;
        private ArrayList<Node<T>> adjacents = new ArrayList<>();
        public Name(T data){
                this.data = data;
        ...
Node<Integer> node1 = new Node<Integer>(10);
Node<String> node2 = new Node<String>(“Apple”);
```

Note

- generic type จะมีกี่ตัวก็ได้ และคั่นด้วย , ex. <T1, T2, D, E>
- generic type จะเป็นอะไรก็ได้ class, interface, abstract class ยกเว้นอย่างเดียวคือ primitive ต้องใช้ Type Wrapper แทน
- obj. ที่สร้างจาก Generic type ต่างกัน แม้เป็นคลาสเดียวกันก็ไม่สามารถใช้แทนกันได้

```
ex.     public void method(A<String> a){ ... }
        A<Integer> a1 = new A<Integer>();
        methodA(a1);            // ERROR
```

- แต่ว่าสามารถสร้าง method ที่ประกาศรับ param แบบไม่ระบุ generic type ได้

```
ex.     Public void method(A a){ ... }
```

Part VI : Comparable

Structure

แบบไม่มี Generic Type

```
public interface Comparable {
        int compareTo(Object o);
}
```

แบบมี Generic Type

```
public interface Comparable<E> {
        int compareTo(E o);
}
```

Implementation

แบบไม่มี Generic Type

```
public class BankAccount implements Comparable {
        public int compareTo(Object other){
                return this.balance – ((BankAccount) other).balance;
```

แบบมี Generic Type

```
public class BankAccount implements Comparable<BankAccount> {
        public int compareTo(BankAccount other) {
                return this.balance – other.balance;
```

Note

- คลาสบางคลาสที่ใช้บ่อยๆ ก้ implements Comparable เช่น Integer, Double, String
- Comparable เก็บอยู่ใน package java.lang
- s1.compareTo(s2)
    - : s1 < s2
    0 : s1 = s2
    + : s1 > s2

Part VII : Comparator

Structure

แบบไม่มี Generic Type

```
public interface Comparator
    int compare(Object o1, Object o2);
```

แบบมี Generic Type

```
public interface Comparator<T>
    int compare(T o1, T o2);
```

Implementation

แบบไม่มี Generic Type

```
public class AccountComparator implements Comparator {
    public int compare(Object o1, Object o2){
        return ((BankAccount) o1).balance – ((BankAccount) o2).balance;
```

แบบมี Generic Type

```
public class AccountComparator implements Comparator<BankAccount> {
    public int compare(BankAccount acc1, BankAccount acc2){
        return acc1.balance – acc2.balance;
```

Note

- การทำงานแบบนี้เป็นแบบ Interface Call Back
- เก็บอยู่ใน package java.util

Part VIII : Cloneable

class Wallet {
private Card card;
private double money;
public Wallet(double money){
this.money = money;
}
}
	Class Card {
	private double money;
	public Card(double money){
	this.money = money;
	}
	}

Solution

step 1 implements Cloneable

* ถ้ามี class อื่นเป็น attribute ต้องทำ clone คลาสนั้นด้วย

class Wallet implements Cloneable {
private Card card;
private double money;
public Wallet(double money){
this.money = money;
}
}
	Class Card implements Cloneable {
	private double money;
	public Card(double money){
	this.money = money;
	}
	}

Step 2 สร้าง method clone และเรียก super.clone();

* super.clone(); มีการ throw CloneNotSupportException ด้วย ดังนั้นตั้ง try-catch รับไว้ด้วย

public Object clone(){
try {
Wallet w = (Wallet) super.clone();
} catch(Exception e) {
return null;
}
}
	public Object clone(){
	try {
	return super.clone();
	} catch(Exception e) {
	return null;
	}
	}

Step 3 clone attribute ที่เป็น referent type ทั้งหมด (นี้เหละเหตุผลที่ให้ทำ clone ทุกคลาส

* ถ้าคลาสไหนที่มี attribute เป็น primitive type ทั้งหมดก็ return ได้เลย ex. Card

```
public Object clone(){
    try {
        Wallet w = (Wallet) super.clone();
        w.card = (Card) this.card.clone();
        return w;
    } catch(Exception e) {
        return null;
    }
}
```

```
public Object clone(){
    try {
        return super.clone();
    } catch(Exception e) {
        return null;
    }
}
```

Warning clone() return ออกมาเป็น Object ดังนั้นเวลาใช้งานอย่าลืม cast ก่อน

Part IX : Boxing & Unboxing

Version java 1.4 or later

Intro

นึกถึงเรื่อง primitive type คงจำกันได้ว่าแต่ล่ะตัวจะมี type wrapper โดยแท้จริงแล้ว ทั้ง 2 อย่างนี้แม้จะเก็บข้อมูลเหมือนกัน แต่ wrapper ก็ยังคงเป็นคลาส ไม่ใช่ primitive เช่น
double กับ Double นั้นไม่ใช่ประเภทเดียวกัน
อ้าว แล้วทำไมทำแบบนี้ได้ล่ะ?

```
Integer i = 3
int j = new Integer(5);
ArrayList<Integer> is = new ArrayList<>();          is.add(3);
```

คำตอบ ก็มัน auto-convert กับ cast แบบ คลาสแม่-คลาสลูกไง
ตอบแบบนี้ ผิดนะครับ อย่าลืมว่า int เป็น primitive จะไปมีคลาสแม่คลาสลูกได้ไง

Auto-Boxing primitive type -> type wrapper

สมัยก่อน (ก่อน 1.4) Integer i = Integer.valueOf(10);

สมัยนี้ (ตั้งแต่ 1.4) Integer i = 10;

Unboxing type wrapper -> primitive type

สมัยก่อน (ก่อน 1.4) int j = i.intValue(); // java ใหม่ ไม่มีเมธอดนี้แล้วนะ

สมัยนี้ (ตั้งแต่ 1.4) int j = i;

Application

เรื่องนี้จะจำเป็นมากๆ ตอนใช้ Generic type เพราะว่า generic ใช้ primitive ไม่ได้ เช่น
ประกาศ ArrayList<Integer> list
ถ้าไม่มี Auto-Boxing จะต้องใช้ list.add(Integer,valueOf(10)); แทน list.add(10);
ถ้าไม่มี Unboxing จะต้องใช้ j = list.get(0).intValue(); แทน j = list.get(0);

Part X : Array

Array Syntax

datatype[] arrName = new dataType[length];

datatype[] arrName = {data1, data2, … };

return new int[] {1, 2, 3}; ~~return {1, 2, 3}~~

Detail

- array เก็บข้อมูลได้เพียงชนิดเดียว แต่สามารถเก็บคลาสลูก ไว้ใน array คลาสแม่ได้
- หมายความว่า ถ้าสร้าง Object[] จะสามารถเก็บข้อมูลได้ทุกอย่าง
- array ก็เป็น object นะ
- array ที่สร้างมาตอนแรก จะใส่ค่า default ของข้อมูลลงไปในทุกๆช่อง
- ขนาดของ array เรียกได้จาก arrName.length เป็น attribute ไม่ใช่ method
- ขนาดของ array จะไม่สามารถเปลี่ยนแปลงได้หลังจากประกาศ ยกเว้นแต่ว่าจะสร้างใหม่

ex. Int[] is = new int[5]; is = new int[7];

- array หลายมิติสามารถประกาศขนาดแค่มิติแรกได้ ex. New int[5][][]

*แต่ไม่สามารถประกาศข้ามมิติว่างได้ ~~ex. New int[5][][5]~~

- สามารถเรียงข้อมูลได้ด้วย Arrays.sort(Object[] a)

* สามารถเรียงค่าของ object ได้ โดยใช้คู่กับ comparator : Arrays.sort(T[] a, Comparator<T> c);

- สามารถคัดลอก array ด้วย Arrays.copryOf(Object[] a, int newLength);

* newLength สามารถใหญ่กว่าเดิมได้ เลยเอาไปใช้เวลาจะขยายขนาดของ array

- Arrays อยู่ใน package java.util

Part XI : Collection คืออะไร

ลักษณะของ collection

<u>แบ่งตามการซ้ำของ element</u>

ซ้ำได้ ex. List, MultiSet

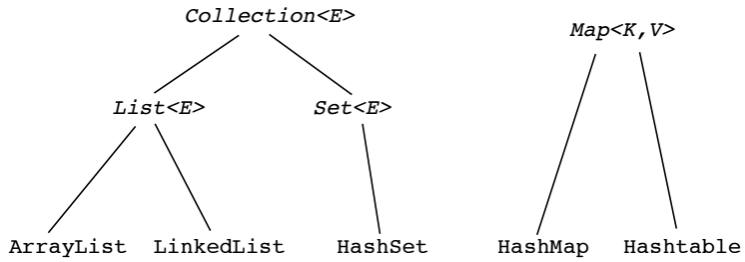
ซ้ำไม่ได้ ex. Set, SortedSet

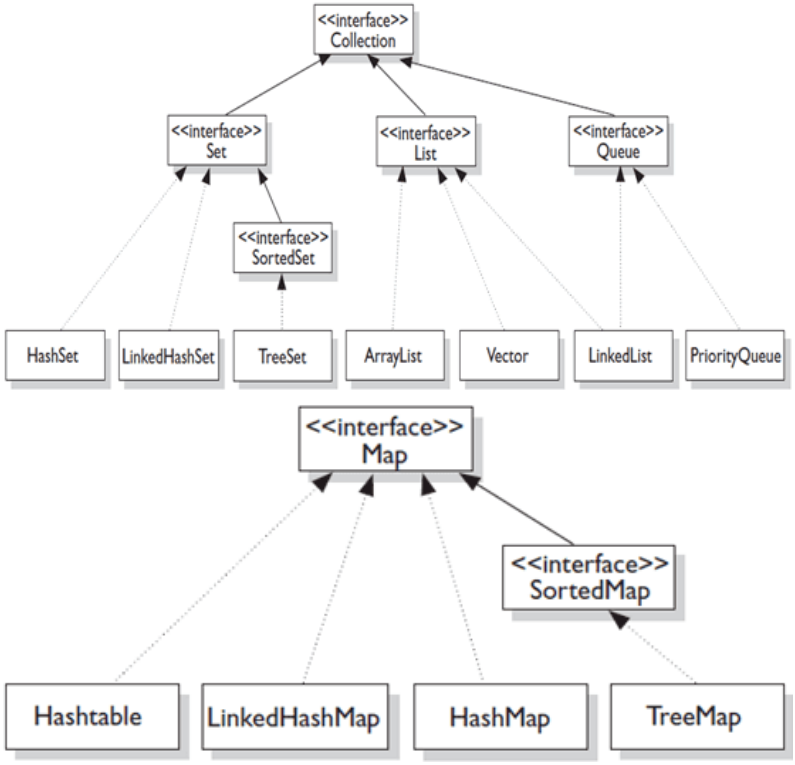
<u>แบ่งตามการเรียง element ตามลำดับการใส่</u>

เรียง ex. List, SortedSet

ไม่เรียง ex. MultiSet, Set

Hierarchy





Interface Collection

package java.util

Interface Collection<E>

Method

boolean add(E o)	เพิ่ม element
boolean contain(Object o)	เช็คว่ามี o อยู่ใน collection มั๊ย
boolean remove(Object o)	ถ้ามี o อยู่จะลบทิ้ง แล้วคืนค่า true ถ้าไม่มีจะคืนค่า false
int size()	คืนค่าขนาดของ collection
void clear()	ล้างค่าข้อมูลทั้งหมด

Interface Iterable

package java.lang

interface Iterable<E>

Method

Iterator<E> iterator()	สร้าง iterator

for-each

```
for(E o: coll) { ... }
```

Interface Iterator

package java.lang

interface Iterator<E>

Method

boolean hasNext()	เช็คว่ายังมีข้อมูลต่อไปอีกมั๊ย
E next()	เลื่อน cursor แล้วดึงข้อมูลออกมา
E remove()	ลบข้อมูลตัวที่ cursor ชี้อยู่

การทำงานของ Iterator

Class ArrayList

package java.util

Detail

- ภายในมีการทำงานแบบ array
- มีเมธอดต่างๆเยอะมาก ช่วยให้ทำงานง่ายกว่าใช้ array
- สามารถเพิ่มขนาดได้เอง

Class LinkedList

package java.util

detail

- ภายในทำงานแบบ linked-list
- implements มาทั้ง List, Queue, Deque(คล้ายๆ stack) ทำให้มีเมธอดทำงานที่หลากหลาย

Class HashSet

package java.util

class HashSet<E> implements Set<E>

detail

- เก็บข้อมูลแบบไม่ซ้ำ และไม่เรียง
- ไม่มีเมธอด get ต้องเข้าถึงโดยใช้ for-each หรือ iterator

hashCode

- class Object จะมีเมธอดตัวนึงชื่อว่า int hashCode()
- เมธอดนี้จะเหมือนเป็นการระบุ id ของ obj.
- ทำให้ set รู้ว่าข้อมูลซ้ำหรือไม่ แล้วก็ยังใช้ตอน == เช็คว่าเป็น obj. ตัวเดียวกันรึเปล่าด้วย

Class TreeSet

detail

- แม้จะเป็น set แต่ก็สามารถ sort ข้อมูลได้
- class ที่เก็บไว้ใน TreeSet จะต้อง implements Comparable

Method

- E first() คืนค่าข้อมูลตัวแรก
- E last() คืนค่าข้อมูลตัวสุดท้าย

Interface Map

package java.util

public interface Map<K, V>

Method

void	clear()	ล้างค่าข้อมูลทั้งหมด
Set<K>	keySet()	คืนค่า set ของกุญแจทั้งหมด
Collection<V>	values()	คืน Collection ของ value ทั้งหมด
Map.Entry<K, V>	entrySet()	คืนค่า set ของ entry ของข้อมูล
V	get(Objext k)	คืนค่า value ที่ตรงกับ key
V	put(K k, V v)	นำ key-value เพิ่มเข้าไป ถ้า key นี้ไม่เคยมีอยู่จะคืนค่า null ถ้ามีแล้วจะคืนค่า value เก่า แล้วเปลี่ยนเป็นค่าใหม่
V	remove(K key)	ลบข้อมูลที่ตรงกับ key และคืนค่าข้อมูลนั้น

Map.Entry

public static interface Entry<K V>

Method

K getKey()	คืนค่า key
V getValue()	คืนค่า value

Detail

- เป็น interface ย่อยที่อยู่ใน Map ตรงนี้ไม่ต้องสนใจมาก รู้แค่ว่ามันจะออกมาจากเมธอด entrySet()
- Map อารมเหมือนกล่องใบใหญ่ ที่เก็บ entry
- แล้ว entry อารมเหมือนถุงที่ใส่ key กับ value แต่ล่ะคู่ไว้ด้วยกัน

การวนลูปใน Map

Map<String, Integer> map = new HashMap<>();

วิธีที่ 1 วนจาก ketSet

```
for(String key : map.keySet())
    sout(map.get(key));
```

วิธีที่ 2 วนจาก values

```
for(Integer value : map.values)
        sout(value);
```

วิธีที่ 3 วนจาก entrySet

```
for(Map.Entry<String, Integer> entry : map.entrySet())
        sout(entry.getKey());
        sout(entry.getValue());
```

HashMap

TreeMap

Detail

- ข้อมูลที่ใส่จะเรียงโดย key
- ดังนั้น key ต้อง implements Comparable

Part XII : Exception

ไม่มีโค้ดตัวอย่างนะ ทำไม่ทัน ไปดูในชีทอาจารย์เอา

Type of Error

Syntax Error เขียนโปรแกรมผิด บิ้มตั้งแต่ยังไม่ได้รัน

Runtime Error รันได้แล้ว แต่พอทำงานไปสักพักบิ้ม

Logical Error รันได้ ไม่บิ้ม แต่คำตอบผิด

Exception

- เป็น object ชนิดนึง ส่วนมากมาจากหลาย class แต่ทั้งหมดมีแม่คือ Exception
- เป็น object ชนิดพิเศษ ที่ทำให้โปรแกรมรู้ว่าเกิดปัญหาขึ้นแล้ว

Error

- เป็น system error หายาก รุนแรง
- ควบคุมไม่ได้ และไม่ควร declare/handle
- สรุป ไม่ต้องไปยุ่งมัน (ชื่อเรื่องก็บอกอยู่ exception จะไปยุ่ง error ทำไม)

Type of Exception

Unchecked Exception

- สืบถอดมาจาก RuntimeException
- ไม่ต้อง declare/handle
- เป็น exception ที่เกิดจาก programmer เอง
- นั้นหมายความว่าให้ไปแก้โค้ดให้มันดีขึ้น ไม่ต้องมานั่ง catch

Checked Exception

- สืบถอดมาจาก Exception
- ต้อง declare/handle
- เป็น exception ที่เกิดจากปัจจัยภายนอก เช่น ผู้ใช้ไม่ยอมกรอกตัวเลข (NumberFormatException)

Throw statement

```
throw new Exception();
throw new Exception("message");
```

- Exception จะเป็นคลาสไหนก็ได้ ขอแค่สืบทอดมาจาก Exception ก็พอ

- ควรมี message เพื่อให้เข้าใจ exception ได้ดีขึ้น
- เมื่อเข้า throw แล้วจะให้ผลเหมือน return คือจะออกจาก function ทันที หรือออกจาก try ทันที

Declare

method(params, ...) throws Exception1, Exception2, ... { }

- ใช้เพื่อบอกว่า method นี้จะเกิด Exception อะไรได้บ้าง
- declare เฉพาะ Checked Exception
- declare เฉพาะ Exception ที่ไม่ได้ catch ไว้
- declare สามารถประกาศเฉพาะคลาสแม่ได้ เช่น FileNotFoundException เป็นลูกของ IOException ก็ประกาศแค่ IOException ได้

Handle

```
try{
    ...
} catch(Exception1 e){ ...
} catch(Exception2 e){ ... }
```

- โปรแกรมจะทำงานใน try ก่อน ถ้าไม่มีปัญหาอะไรก้จบ try ไป
- ถ้าเกิดใน try บึ้มขึ้นมา จะออกแล้วไปหา catch ที่ใช่ เข้าทันที

<u>try block</u>

- เป็นโค้ดส่วนที่มีโอกาสเกิด Exception
- Exception ในนี้อาจเกิดจาก method อื่นที่เรียกในนี้ หรือ throw เองใน try ก็ได้

<u>catch block</u>

- ในวงเล็บใส่ประเภทของ Exception
- ควรเอา subclass ขึ้นก่อน superclass

สรุป throw / declare / handle

- การ throw คือการที่แสดงให้ระบบรู้ว่าตรงนี้มีปัญหาแล้วนะ
- handle คือการที่จะจัดการกับปัญหานั้น เช่น แสดงผลออกมา เด้งป๊อปอัพ ปิดโปรแกรม
- declare บางเมธอดก็ไม่สมควรจัดการกับ exception เอง เลยต้องทำการส่งไปให้ caller เช่น สมมติ views เรียก method deposit ใน models

แล้ว deposit เกิด เงินติดลบException ขึ้นมา

ล่ะคือจะให้ catch ใน deposit แล้วเด้งหน้าต่างเตือนขึ้นมา มันก้ไม่ใช่ไง

เพราะ models ไม่ควรแสดงผล

ทีนี้ก้เลยมี declare เอาไว้ให้แบบ ถ้าเกิด exception นี้ขึ้นมาล่ะทำอะไรกับมันไม่ได้ งั้นก้โยนมันออกไปเลย โยนแล้วไปไหน? Exception ที่โยนจาก declare จะไปยังผู้เรียก (caller) ในที่นี้ก็คือ views ซึ่ง views รู้วิธีรับมือกับปัญหาเงินติดลบนะ เดี๋ยวจะเด้งป๊อปอัพให้ views ก้ตั้ง catch รอไว้ จบ

Class Throwable

- เป็นแม่ของ Exception และ Error

<u>method</u>

String	getMessage();	แสดง message ที่ใส่ไปตอนสร้าง exception
void	printStackTrace();	แสดง error
String	toString();	

Finally Block

- ใช้ร่วมกับ try/catch
- ใช้เพื่อ clean up ระบบ เช่น ปิด file
- จะถูกรันเสมอ ไม่ว่าจะจบงานที่ try หรือ catch
- ถ้ามี return จะมาเข้า finally ก่อน return
- ถ้าใน catch มี throw ก็จะมาเข้า finally ก่อน
- แต่ถ้าไปเจอ System.exit() แล้ว จะไม่เข้า finally แล้วนะ เพราะโปรแกรมปิดไปล่ะ

สร้าง Exception ใช้เอง

- unchecked ให้ extends RuntimeException

- checked ให้ extends Exception

ภายในทำแค่ 2 คอนสตรัคเตอร์

```
Public class NewException extends Exception {
        public NewException(){ }
        public NewException(String message){
            super(message);
        }
}
```

Part XIII : IO

IO API (คลาสที่เกี่ยวกับ Input/Output)

package java.io

แบ่งออกเป็น 2 สาย

สาย Input มีเมธอด int read()

abstract class InputStream

abstract class Reader

สาย Output มีเมธอด void write(int)

abstract class OutputStream

abstract class Writer

InputStream/OutputStream และ Reader/Writer

InputStream/OutputStream

- อ่าน/เขียน byte เหมาะกับไฟล์ต่างๆที่ไม่ใช่ text
- System.in เป็น InputStream , System.out/err เป็น PrintStream

Reader/Writer

- อ่าน/เขียน ตัวอักษร เหมาะกับไฟล์ text

InnputStreamReader/OutputStreamWriter

- ใช้แปลงจาก Stream เป็น Reader/Writer
- เป็น subclass ของ Reader/Writer

การอ่านไฟล์

FileReader

- เป็น subclass ของ Reader

FileReader reader = new FileReader(String filename)

* constructor จอง FileReader จะโยน **FileNotFoundException** ด้วยเวลาสร้าง obj

method

- int read() ตัวนี้ได้มาจาก Reader

BufferedReader

- ทำงานได้เร็วกว่า FileReader

- มีเมธอด readLine ทำให้อ่านได้ทีล่ะบรรทัด

BufferedReader buffer = new BufferedReader(Reader reader)

- BufferedReader สร้างจาก Reader สามารถใส่อะไรก้ได้ เช่น

new BufferedReader(new FileReader("file1.txt")) ใส่ file reader

new BufferedReader(new InputStreamReader(System.in)) ใส่ input stream จาก console

method

- String readLine() อ่านทั้งบรรทัด

Note read() และ readLine() โยน IOException ทั้งคู่ อย่าลืม catch

การปิดไฟล์

void close() ปิดไฟล์

เมธอดนี้เป็นของ Reader ดังนั้นจะมีทั้งใน FileReader และ BufferedReader

จะเลือก buffer.close() หรือ reader.close() มีค่าเท่ากัน เลือกสักอย่าง

Note การลืมปิดไฟล์ ไม่เกิด Error ไม่มีอะไรแจ้งบอกด้วย แต่จะมีปัญหา เพราะโปรแกรมจะครองไฟล์ไว้คนเดียวจนกว่าจะตาย โปรแกรมอื่นจะเข้าถึงไฟล์นี้ไม่ได้

Note ดังนั้นการปิดไฟล์จำเป็นมาก จึงควรไว้ที่ finally

Note close() จะโยน IOException อย่าลืม catch

สรุปโค้ดอ่านไฟล์

```
...
FileReader reader = null;
try {
        reader = new FileReader(filename);
        BufferedReader buffer = new BufferedReader(reader);
        ...
        String line = null
        while ( (line = buffer.readLine()) != null)
                ...
        ...
} catch(FileNotFoundException e) {
        ...
} catch(IOException e) {
        ...
} finally {
        try {
                if(reader != null)
                        reader.close();
        catch (IOException e) {
                ...
        }
}
```

การเขียนไฟล์

FileWriter ใช้แค่เปิดไฟล์ (จริงๆจะเขียนไฟล์ก้ได้ แต่ลำบาก)

FileWriter writer = new FileWriter(filename)

แบบที่ 1 ใช้ PrintWriter

- ใช้ง่าย มี println(...)

PrintWriter out = new PrintWriter(writer)

แบบที่ 2 ใช้ BufferedWriter

- ทำงานเร็วกว่า แต่ใช้ยากเพราะ ไม่มี println(...)

BufferedWriter out = new BufferedWriter(writer)

- เวลาเขียนใช้ out.write(...) เวลาจะขึ้นบรรทัดใหม่ใช้ out.newLine() แทน “\n”

แบบที่ 3 ใช้ ทั้ง 2 ตัวเลย

PrintWriter out = new PrintWriter(BufferedWriter(writer))

การ flush

- ทุกครั้งที่เขียนไฟล์(จะ println หรือ write ก้ได้) บางครั้งเขียนไปแล้วระบบยังไม่ได้เซงลงไฟล์ให้ บางทีก้อาจเขียนไม่ติด

- ดังนั้นเวลาที่เขียนไฟล์เสร็จแล้ว หรือตอนไหนก็ได้ที่อยากเมคชัวร์ว่าจะเซฟลงไฟล์ ให้เรียน out.flush()

การเขียนทับ/เขียนต่อ

<u>เขียนทับ</u>

FileWriter writer = new FileWriter(filename) หรือ new FileWriter(filename, false)

<u>เขียนต่อ</u>

FileWriter writer = new FileWriter(filename, true)

สรุปโค้ดเขียนไฟล์

```
...
FileWriter writer = null;
try {
        writer = new FileWriter(filename);
        PrintWriter out = new PrintWriter(writer);
        ...
        out.println(...);
        ...
        out.flush();
 } catch (FileNotFoundExcaption e) {
        ...
} catch (IOException e) {
        ...
} finally {
        try {
                if (writer != null)
                        writer.close()
        } catch (IOException e) {
                ...
```

Scanner

package java.util

Constructor : Scaner(InputStream), Scanner(Reader), Scanner(File)

ex. Scanner sc = new Scanner(System.in);

Scanner sc = new Scanner(new FileReader(filename));

Scanner sc = new Scanner(new File(filename));

method

String nextLine() อ่านไฟล์จนเจอ \n (อ่านทั้งบรรทัด)

String next() อ่านไฟล์จนเจอเว้นวรรค

int nextInt() อ่านไฟล์แล้วแปลงเป็น int ให้ (มีทั้ง nextDouble nextBoolean, ...)

Boolean hasNext() ตรวจสอบว่ามีค่าให้อ่านอีกมั๊ย

Boolean hasNextInt() ตรวจสอบว่ามีตัวเลขต่ออีกมั๊ย (มี hasNextDouble, ... ด้วยเหมือนกัน)

Note ถ้าข้อมูลที่อ่านมาเป็น String แต่ใช้ nextInt จะได้ InputMismatchException

Note ถ้าข้อมูลหมดแล้วจะอ่านต่อจะได้ NoSuchElementException

เขียน/อ่าน Object ลงไฟล์

* ไฟล์ที่จะใช้นามสกุล .ser

* class ที่จะเอามาเขียนได้ต้อง implements Serializable

เขียน object ลงไฟล์

```
try {
        FileOutputStream fileStream = new FileOutputStream(filename);     // FileNotEx
        ObjectOutputStream os = new ObjectOutputStream(fileStream);     // IOEx
        os.writeObject(a1);         // IOEx
        ...
        os.flush();         // IOEx
} catch (FileNotFoundException e) { ... }
catch (IOException e) { ... }
finally {
        os.close();
}
```

อ่าน obj จากไฟล์

```
try {
        FileInputStream fileStream = new FileInputStream(filename);        // FileNotEx
        ObjectInputStream os = new OutputStream(fileStream);               // IOEx
        Object o1 = os.readObject();        // ClassNotFoundEx
        A a2 = (A) o1;
 } catch (FileNotFoundException e) { ... }
catch (IOException e) { ... }
catch (ClassNotFoundException e) { ... }
finally {
        os.close();
}
```

Note ถ้าไฟล์ที่อ่านจบแล้ว แต่ยังอ่านต่อจะเกิด EOFException

Note InvalidClassException

```
private static final long serialVersionUID = 4078489481044090127L;
```

Part XIV : Multithread

รู้จักกับ Thread

สมมติว่าให้ Thread เป็นเหมือนคน 1 คนที่ทำงานได้ ทุกวันนี้ที่เราเขียนโปรแกรมก็เหมือนกับการให้คน 1 คนทำงานคนเดียวจนเสร็จ สมมติอีกว่า Thread คนเดียวที่เราใช้อยู่นี้ชื่อว่า Main

ทีนี้ต่อมาคนๆเดียวของเราทำงานคนเดียวก็เหนื่อย เลยคิดที่จะหาคนอื่นมาช่วย แต่ก่อนที่จะให้คนอื่นมาช่วยนั้น ขั้นแรกต้องเตรียมของที่จะใช้ทำงาน และวิธีการทำงานไว้ให้ (Task) จากนั้นจึงแจกให้กับ Thread คนอื่นๆ และ Main ก็จะนั่งคอยให้คนอื่นๆทำงาน

Let's Do It

ex. ใช้ multithread ในการอ่านไฟล์ลงไปใน obj. Data

step 1 สร้าง Task ที่จะแจกให้แต่ล่ะ thread

```
public class FileReaderTask implements Runnable
        private String filename;
        private Data data;
        public FileReaderTask(String filename, Data data)
                this.filename = filename
                this.data = data
        @Override
        public void run()
                ...
                data.add(buffer.readline())
                ...
```

- ขั้นแรกในการจะสร้าง Task ต้องคิดก่อนว่างานย่อยๆที่เราแบ่งมาให้ Thread ทำนั้น ต้องใช้อะไรบ้างในการทำให้เสร็จ
- นำของที่จำเป็นเก็บไว้เป็น attribute และรับมาผ่านทาง constructor
- งานที่ให้เธรดทำจะอยู่ run

step 2 สร้าง Thread และแจกจ่ายงาน

```
public void main(String[] arg)
        ...
#1      String[] filenames = {...};
        Data[] datas = new Data[filenames.lenght];
        Thread threads = new Thread[filenames.lenght];
        for(int i=0; i<filenames.lenght; i++)
                datas[i] = new Data();
#2              FileReaderTask task = new FileReaderTask(filename, datas[i]);
#3              threads[i] = new Thread(task);
#4              threads[i].start();
#5      int n = 0;
        while(n != threads.lenght)
                n = 0
                for(Thread t : threads)
                        if(!t.isAlive())
                                n++;
                Thread.sleep(1000);
#6      ...
```

#1 เตรียมของทุกอย่างให้พร้อม สร้าง array สำหรับเก็บข้อมูลและ thread

#2 สร้าง task ขึ้นมาสำหรับ thread

#3 สร้าง thread ขึ้นมา แล้วส่ง task ไปให้

#4 ให้ thread เริ่มทำงาน

#5 นั่งรอ thread ทำงานเสร็จ อาจมี sleep ด้วย เพื่อไม่ให้ต้องถามตลอด

#6 หลังจาก thread ทุกตัวทำงานเสร็จแล้ว ก็ไปทำงานอย่างอื่นต่อ

Thread.sleep

สร้างสามารถหยุดการทำงานของเธรดชั่วคราวได้ด้วย Thread.sleep(millisec)

เมธอดนี้จะโยน InterruptException อย่าลืม catch

Terminating Thread - หยุดการทำงานของ Thread มี 2 วิธี

- thread.stop() สั่งให้หยุดทันที แบบนี้จะไม่ค่อยดี
- thread.interrupt() บอกว่าให้หยุด แบบนี้เธรดยังพอมีเวลาเคลียข้อมูล หรือปิดไฟล์ได้

Thread.interupt() : boolean คอยเช็คเมธอดนี้ใน run() ถ้ามีค่าเป็นจริงให้เคลียร์ข้อมูล

* เมื่อ Thread.sleep() อยู่ แต่โดน interrupt() จะเกิด InterruptException

Race Condition

ปรากฏการที่มีหลายเธรดพยายามแย่งกันแก้ไขข้อมูลชุดเดียวกัน ทำให้ผลลัพธ์มรความผิดพลาด

ex. การ deposit และ withdraw โดยใช้ multithread ที่อาจารย์ทำให้ดูในห้อง

Lock Object

วิธีแก้ปัญหา Race Condition โดยการจำกัดแค่ Thread เดียวในการแก้ไขข้อมูลในเวลาเดียวกัน

step 1 สร้าง Lock object ไว้ใน class ที่จะมีการแก้ไขข้อมูล

```
public class BankAccount
        private double balance
        private Lock balanceLocker = new ReentrantLock();
        ...
        public void deposit(double amt)
                balanceLocker.lock();
                balance += amt;
                balanceLocker.unlock();
        public void witdraw(double amt)
                balanceLockrt.lock();
                balance -= amt;
                balanceLocker.unlock();
```

การทำงาน

Lock เป็น obj. ชนิดพิเศษ มันจะจดจำว่า Thread ตัวไหนเป็นคน lock มัน และมันจะไม่ยอมให้เธรดอื่น lock หรือ unlock นอกจากเธรดตัวนั้น ถ้าเธรดอื่นพยายามจะ lock ซ้ำ เธรดนั้นจะถูก sleep

ex. Thread deposit1 เป็นคน lock ขณะกำลังฝากเงินอยู่ Thread deposit2 เดินเข้ามาจะขอ lock บ้าง ตัว Lock obj. จะสั่งให้ deposit2 ไปนอน

Dead Lock

ปรากฏการณ์ที่ Thread ตั้งแต่ 2 Thread ขึ้นไป ต่างรอซึ่งกันและกัน

ex. Thread withdraw1 จะถอนเงิน แต่เงินไม่พอให้ถอน ก็เลยจะยืนรอให้มีคนมาฝากเงิน แล้ว Thread deposit1 จะเอาเงินมาฝาก แต่ว่า withdraw1 ยัง lock ไว้อยู่เลยไม่สามารถไปฝากได้ withdraw1 ก็รอไปเหอะแต่ไม่ยอมปล่อย lock deposit1 ก็รอเมื่อไหร่จะ unlock

Condition Object

```
public class BankAccount
        private Lock balanceLocker;
        private Condition sufficientFundsCondition;
        public BankAccount()
                balanceLocker = new ReentrantLock()
                sufficientFundsCondition; = balanceLock.newCondition()
        public void deposit(double amt)
                balanceLocker.lock();
                balance += amt
                sufficientFundsCondition.signalAll();
                balanceLocker.unlock();
        public void withdraw(double amt) throws InterruptException
                balanceLocker.lock()
                while(balance < amt)
                        sufficientFundsCondition.await();
                balance -= amt
                balance.unlock()
```

การทำงาน

- สร้าง condition มาจาก lock
- เมื่อเรียก condition.await() จะทำให้ Thread ไปนั่งพักจิบชา รอเงินเข้า
- await() จะ throws InterruptException ด้วย
- ส่วนใน deposit จะเรียก signalAll เพื่อบอกเธรดที่นั่งจิบชารออยู่ว่าเงินเข้าแล้วนะ

Note

- unlock() ควรทำใน finally
- multithread เข้าใจยาก อะไรจำได้ก้จำๆไปล่ะกัน

Part XV : Design Pattern

Design Pattern

คือรูปแบบในการเขียนโปรแกรมที่ดี ที่มีคนออกแบบมาแล้วสำหรับแก้ไขปัญหาต่างๆ หรือสอดคล้องกับ Principle ต่างๆ

Principle บางตัวที่น่าสนใจ

Principle คือ หลักการในการเขียนโปรแกรมที่ดี

KISS Principle (Keep It Simple, Stupid)

- การออกแบบโค้ดต้องทำให้ง่ายต่อการใช้งาน
- อย่าเขียนโปรแกรม(หรือฟังก์ชั่น) ที่ใช้งานยาก (น่าจะรวมถึงฟังก์ชั่นที่มีพารามิเตอร์เยอะๆด้วย)

DRY Principle (Do not Repeat Yourself)

- อย่า copy and paste
- ไม่ควรมีโค้ดที่หน้าตาเหมือนกันหลายๆที่ในโปรแกรม

Loosely Coupled

- class มีความสัมพันธ์กันค่อนข้างน้อย
- โค้ดจะยืดหยุ่น แก้ไขคลาสนึงจะกระทบกับอีกคลาสไม่มาก
- เพิ่ม feature ได้โดยไม่ต้องแก้ไขโค้ดเก่า

Height Cohesion

- คลาสนึงๆ ควรทำงานเฉพาะอย่าง ไม่ควรสร้างคลาสที่ทำงานแทบทุกอย่าง

OO Basics

Abstraction – การใช้งาน class หรือ method ต่างๆได้ โดยไม่ต้องรู้กลไกการทำงานของมัน

Encapsulation – การรวบรวม method, attribute เข้าไว้ด้วยกัน

Polymorphism – การที่ code เหมือนกัน แต่ทำงานต่างกัน ขึ้นอยู่กับ subclass ต่างๆ

Inheritance – การสืบทอดคุณสมบัติ

Part XV : Design Pattern – Strategy Pattern

ที่มาของปัญหา

สมมติมีคลาส A กับ B ที่ว่างข้างๆเผื่อวาดรูป

คลาสทั้งสองมีเมธอด show และ run เหมือนกัน

show จะทำงานเหมือนกันทั้งสองคลาส

แต่ run จะทำงานต่างกัน

<u>การออกแบบแบบเก่า</u>

โอเค ถ้า show มันเหมือนกันงั้นสร้าง superclass แล้วเขียน show ไว้ในนั้นดีกว่า

```
public abstract class O
        public void show()
                /* show method */
        public abstract void run();

public class A extends O
        public void run()
                /* run A */
public class B extends O
        public void run()
                /* run B */
```

เสร็จเรียบร้อย แต่แล้วต่อมาวันนึงโปรแกรมโตขึ้น ต้องการเพิ่ม class C / D ที่ run แบบB / E ที่ run แบบA

```
public class C extends O
        public void run()
                /* run C */
public class D extends O
        public void run()
                /* run B */
public class E extends O
        public void run()
                /* run A */
```

จะเห็นว่ามีโค้ดที่เหมือนกันอยู่หลายที่มาก ซึ่งขัดกับ DRY Principle

แก้ปัญหาด้วย Strategy Pattern

step 1 แยกเมธอดที่ทำงานซ้ำกันออกมาเป็น interface ใหม่

```
public interface RunBehavior
        void run();
```

step 2 สร้างคลาสใหม่เพื่อเก็บการทำงานแบบต่างๆโดย implements จาก step 1

```
public class ARunBehavior implements RunBehavior
        public void run()
                /* run A */
public class BRunBehavior implements RunBehavior
        public void run()
                /* run B */
public class CRunBehavior implements RunBehavior
        public void run()
                /* run C */
```

step 3 กลับไปแก้คลาสแม่ (class O)

- เพิ่ม attribute ใหม่ โดยมี static type เป็น interface ที่สร้างไว้
- แก้ไขเมธอดที่มีปัญหาในตอนแรก (method run ใน O) โดยให้เรียกใช้เมธอดที่สร้างไว้ใน interface
- เพิ่มเติม setter ได้ เพื่อให้ obj สร้างสามารถเปลี่ยนแปลงการทำงานได้

* setter จะทำหรือไม่ขึ้นอยู่กับความต้องการของโปรแกรมว่า obj สามารถเปลี่ยนการทำงานได้หรือไม่

```
public class abstract O
        protected RunBehavior runBehavior;
        public void show()
                /* show */
        public void run()
                runBehavior.run()
        public void setRunBehavior(RunBehavior runBehavior)
                this.runBehavior = runBehavior
```

step 4 คลาสลูกเพิ่มเติมเลือก method ที่ต้องการจะใช้ใน constructor

```
public class A extends O
        public A()
                runBahavior = new ARunBehavior();
public class B extends O
        public B()
                runBehavior = new BRunBehavior();
public class C extends O
        public C()
                runBehavior = new CRunBehavior();
public class D extends O
        public D()
                runBehavior = new BRunBehavior();
public class E extends O
        public E()
                runBehavior = new ARunBehavior();
```

เวลาเรียกใช้

```
public void main(String[] arg)
        A a1 = new A();
        B b1 = new B();
        a1.run();           // เรียกใช้ปกติไม่ต้องสนใจอะไร
        b1.run();
        b1.setRunBehavior(new CRunBehavior());    // สามารถเปลี่ยนการทำงานได้
```

Note

- Strategy ไม่ได้จำเป็นต้องมีคลาสแม่เสมอ อาจเป็นคลาสๆเดียว ที่สามารถเปลี่ยนการทำงานได้ก็ทำได้เช่นกัน ex. CashRegister
- Strategy เหมาะกับคลาสที่มีการทำงานได้หลายแบบ หรือคลาสที่สามารถเปลี่ยนเมธอดได้

Part XV : Design Pattern – Observer Pattern

ที่มาของปัญหา

- มีคลาสนึงที่มีข้อมูลเปลี่ยนแปลงตลอดเวลา และมีอีกคลาสนึงที่จะคอยเอาข้อมูลนั้นไปใช้ตลอดเวลาเช่นกัน

Observer แบบสร้างเอง

ข้อดี

- ถ้าสร้างเองจะเป็น interface ทำให้ใช้ง่ายได้คล่องกว่า

-

ข้อเสีย

- ก็ต้องสร้างเองไงเสียเวลาสร้าง เสียเวลามานั่ง implements อีก

Observer แบบใช้ของที่ java มีให้

ข้อดี

- ไม่ต้องเสียเวลามาสร้างเอง แค่ extends มาก็พอ

ข้อเสีย

- java ฟิกพารามิตเตอร์ที่ observable จะส่งไปให้ observer ไว้แล้ว ทำให้บางทีก็ไม่ค่อยยืดหยุ่นเท่าไหร่
- ต้อง extends Observable ทำให้การใช้งานค่อนข้างจำกัดกว่า

Observer แบบสร้างเอง

step 1 สร้าง interface Subject และ Observer ดังนี้

```
public interface Subject
        void registerObserver(Observer o);
        void removeObserver(Observer o);
        void notifyObservers();
public interface Observer
        void update(...);
```

* ใน Subject จะให้ import Observer ระวังอย่า import Observer ที่มาจากจาว่า

* พารามิเตอร์ของ update แล้วแต่ว่าจะใส่อะไร เช่น

update(double temp, double humidity) ใส่เฉพาะค่าที่จำเป็นไปเลยก็ได้

update(Subject subject) หรือใส่ตัว subject ไปเลยก็ได้แล้วค่อยไป get ค่าข้างใน

step 2 implements Subject ไปยังคลาสที่มีข้อมูลเปลี่ยนแปลงตลอดเวลา

```
public class WeatherData implements Subject
        private ArrayList<Observer> observers = new ArrayList<>();
        ...
        public void registerObserver(Observer o)
                observers.add(o)
        public void removeObserver(Observer o)
                observers.remove(o)
        public void notifyObservers()
                for(Obsrver o : observers)
                        o.update(temperature)
        public void setTemperature(double temp)
                this.temperature = temp
                notifyObservers()
```

* จุดที่จะเรียกใช้เมธอด notifyObservers() คือหลังจากที่ข้อมูลภายในคลาสเปลี่ยนแปลง จะไปตรงไหนก้ได้ที่ข้อมูลเปลี่ยน จะต้องเรียก notify

step 3 implements Observer ไปยังคลาสที่ต้องการข้อมูล

```
public class CurrentTempDisplay implements Observer
        ...
        public void update(double temp)
                sout(“Current temp “ + temp)
public class TempFileWriter implements Observer
        PrintWriter tempPrinter;
        ...
        public void update(double temp)
                tempPrint.println(temp)
public class TempStat implement Observer
        ArrayList<Double> tempData;
        ...
        public void update(double temp)
                tempData.add(temp)
```

* Observer แต่ล่ะตัวอาจเอาข้อมูลไปทำอะไรก็ได้ ถ้าทำ mvc ก็จะต้องเลือกแพคเกจดีๆ

CurrentTempData ทำหน้าที่แสดงผล ควรจะอยู่ใน views

TempFileWriter ทำหน้าที่เขียนไฟล์ ควรจะอยู่ใน controllers เพราะติดต่อ IO

TempStat ทำหน้าที่เก็บสถิติอุณหภูมิ ควรจะอยู่ใน models เพราะทำหน้าที่เก็บข้อมูล

ส่วน interface Observer นั้นจะเก็บไว้ที่ models ก็ได้แล้วแต่

step 4 นำ subject และ observer ไปใช้

```
public static void main(String[] args)
        /* สร้าง subject และ observer */
        WeatherData weather = new WeatherData();
        CurrentTempDisplay current = new CurrentTempData();
        TempFileWriter file = new TempFileWriter();
        TempStat stat = new TempStat();
        /* ต่อด้านขวา ที่เต็ม */
```

```
        /* register Observer */
        weather.registerObserver(current)
        weather.registerObserver(file)
        weather.registerObserver(stat)
/* เปลี่ยนค่าจะ update ไปยัง Observer ทุกตัว */
        weather.setTemperature(25.7)
```

Observer แบบที่ java มีมาให้

step 1 extends Observable ก่อน

```
public class WeatherData extends Observable
        ...
        public void setTemperature(double temp)
                this.temperature = temp
                setChanged()
                notifyObservers()
```

- จะเห็นว่าเราไม่ต้อง Override เมธอดอะไรเลย เพราะ Observable เขียนไว้ให้หมดล่ะ
- เมธอดที่มีมาให้
    - void addObserver(Observer o) เพิ่ม Observer ตัวใหม่
    - void deleteObserver(Observer o) ลบ Observer
    - void setChanged() เรียกใช้ก่อน notify ทุกครั้ง
    - void notifyObservers() notify Observer ทุกตัว

* notifyObservers สามารถใส่พารามิเตอร์อะไรก็ได้เพิ่มได้ 1 ตัว ex. notifyObservers(“hello”)

step 2 implements Observer ไปยังคลาสที่ต้องการข้อมูลแบบนี้

```
public class CurrentTemp implements Observer
        private Observable observable
        public CurrentTemp(Observable observable)
                this.observable = observable
                observable.addObserver(this)
        public void update(Observable o, Object arg)
                if(o instanceof WeatherData)
                        WeatherData weather = (WeatherData) o;
                        sout(weather.getTemperature())
```

* add ตัวเองให้กับ observable ใน constructor

* update จะรับพารามิตเตอร์ 2 ตัว

- o : คือ Observable ผู้ที่ notify
- arg : คือค่าที่ใส่เพิ่มมาใน notifyObserver()

ถ้าไม่ใส่ arg จะเป็น null

ถ้าใส่ arg จะมี type เป็น Object ต้อง cast ก่อนใช้นะ

step 3 เอาไปใช้

```
public void main(String[] args)
        WeatherData weather = new WeatherData()
        CurrentTemp current = new CurrentTemp(weather)
        ...
        weather.setTemperature(25)
```

Part XV : Design Pattern – SOLID

SOLID คือหลักธรรม 5 ข้อในการเขียนโปรแกรมที่ดีประกอบด้วย

Single Responsibility Principle : SRP

- class หนึ่งๆ ควรจะมีหน้าที่เพียงหน้าที่เดียวเท่านั้น

- วิธีเช็คว่าคลาสนั้นเป็น SRP หรือไม่ ทำแบบนี้

The *classname method* itself.

เติมชื่อคลาสและชื่อเมธอดลงไป จากนั้นดูว่าสมเหตุสมผลไหม

ex. The car drive itself. รถขับตัวเอง? อันนี้ไม่ใช่ล่ะ

The car start itself. รถเริ่มตัวเอง แบบนี้อ่ะพอได้

Open/Closed Principle : OCP

- Open for extension : ออกแบบคลาสให้มีโอกาสเพิ่ม feature ใหม่ๆได้

- Close for modify : แต่ออกแบบคลาสเพื่อไม่ให้ในอนาคตจะเพิ่มอะไร ไม่ต้องมาแก้โครงสร้างของเก่า

Liskov Substitution Principle : LSP

- subtype จะต้องสามารถทำงานแทน supertype ได้เสมอ

* ใช้แทนในที่นี้ ต้องหมายความว่าทำงานได้ดีด้วย

Delegation – การแบ่งการทำงานไปให้คลาสอื่นช่วยทำ

Interface Segregation Principle : ISP

- แบ่งแยกเมธอดต่างๆออกมาเป็น interface แล้ว implement เอา

- คลาสที่จะเรียกใช้ก็เรียกใช้ interface เพื่อลด coupling

Dependency Inversion Principle : DIP

- เวลาเขียนโปรแกรมใช้ static type เป็น supertype ที่สูงที่สุดเท่าที่จะใช้งานได้

Refactoring – การปรับ design ของโปรแกรม โดยไม่ทำให้การทำงานเปลี่ยนไป

Class Smell

- feature envy : ใช้ method ของคลาสอื่นบ่อยเกินไป

- refused bequest : คลาสลูก override การทำงานขัดกับคลาสแม่

- freeloader/lazy class : class ที่มีหน้าที่น้อยเกินไป

- source code ซ้ำซ้อน (ขัดกับ DRY) / method หรือ class ที่มีหน้าที่ใหญ่เกินไป (ขัดกับ SRP)

Appendix A : Package

java.awt	**java.util**	**javax.swing**
classes	interfaces	classes
Color	Collection	BorderFactory
Dimension	Comparator	BoxLayout
Font	Iterator	ButtonGroup
Graphics	List	JFrame, JButton, …
Graphics2D	Set	ตระกูล J ทุกอย่าง
GridLayout	Map	Timer
BorderLayout	classes	**java.lang**
FlowLayout	ArrayList	interfaces
Polygon	Arrays	Cloneable
Rectangle	Collections	Comparable
พวก Label TextField	HashMap	Iterable
ที่ไม่มี J ก็อยู่ในนี้	HashSet	Runnable
java.awt.event	LinkedList	classes
interfaces	TreeMap	Type Wrapper ทั้ง 8 + String
ActionListener	TreeSet	Math
อันอื่นๆที่ลงท้ายด้วย Listener	**java.util.concurrent.locks**	Object
classes	interfaces	System
ActionEvent	Lock	Thread
อันอื่นๆที่ลงท้ายด้วย Event	classes	
java.awt.geom	ReentrantLock	
classes	**java.io**	
AffineTransform	interfaces	
GeneralPath	Serializable	
Ellipse2D	classes	
	InputStream	
	OutputStream	
	อื่นๆคงพอเดาๆได้	

Appendix B : Exception

Exception	method
CloneNotSupportException	Super.clone()
IOException	.close() .read() .readline() .flush() .write(...)
FileNotFoundException	new FileReader(filename) new FileWriter(filename) new FileInputStream(filename) new FileOutputStream(filename) new File(filename)
ClassNotFoundException	.readObject()
InterruptException	Thread.sleep(millisec)
NullPointerException	ใช้ null
ClassCastException	Cast คลาสที่ไม่เกี่ยวกัน
IndexOutOfBoundsException	Index เกิน
ArithmeticException	หารด้วย 0

Unchecked Exception

Checked Exception