บทที่ 4
พื้นฐานจาวาคอลเลคชั่น

จาวาคอลเลคชั่นคือชุดของอินเตอร์เฟสและคลาสในแพคเกจ java.util คอลเลคชั่นหนึ่งนั้นคือออบเจ็กต์ที่ประกอบขึ้นมาจากสมาชิกย่อย (ซึ่งสมาชิกย่อยนั้นจะเป็นข้อมูลพื้นฐานหรือเป็นพอยต์เตอร์ไปยังออบเจ็กต์ก็ได้) ตัวอย่างเช่น อาร์เรย์ ซึ่งถือเป็นคอลเลคชั่นของสิ่งชนิดเดียวกัน

คอลเลคชั่นคลาส คือ คลาสที่ออบเจ็กต์ของมันประกอบขึ้นมาจากสมาชิกย่อย ซึ่งสมาชิกย่อยนั้นต้องเป็นพอยต์เตอร์ไปยังออบเจ็กต์ ดังนั้นเราไม่สามารถใช้ชุดข้อมูลพื้นฐานเป็นส่วนประกอบของคอลเลคชั่นคลาสได้ (แต่อย่าลืมว่าเรามี wrapper class ที่ใช้แทนชุดข้อมูลพื้นฐานได้) ในการสร้างออบเจ็กต์ของคลาส Integer ขึ้นมาจากจำนวนเต็ม j เราใช้โค้ดได้ดังนี้

```
Integer myInteger = new Integer(j);
```

โครงสร้างการเก็บข้อมูลของคอลเลคชั่นคลาส

วิธีที่ง่ายที่สุดคือเก็บสมาชิกของคอลเลคชั่นออบเจ็กต์ไว้ในอาร์เรย์ ซึ่งเราอาจให้อาร์เรย์เป็นหนึ่งฟิลด์ของออบเจ็กต์นั้น คลาสแบบนี้ถือเป็นคอลเลคชั่นคลาสแบบเก็บข้อมูลติดกัน (contiguous - collection class) จาวามีคลาสแบบนี้คือ ArrayList

แต่ว่าการใช้อาร์เรย์ หรือออบเจ็กต์ภายในคอลเล็คชั่นออบเจ็กต์นั้น ผู้ใช้เพียงแต่เรียกใช้เมธอดให้เหมาะสมเท่านั้นก็พอ ไม่จำเป็นต้องรู้ถึงการทำงานภายใน ดังนั้นจึงไม่จำเป็นต้องรู้ว่าภายในคอลเลคชั่นออบเจ็กต์นั้นมีอาร์เรย์อยู่หรือไม่ จึงเป็นหน้าที่ของผู้เขียนคอลเลคชั่นคลาสที่จะเขียนโค้ดเตรียมไว้จัดการกับการขยายอาร์เรย์เอง

จาวามีโครงสร้างข้อมูลที่ใช้กันบ่อยๆเขียนเป็นคอลเลคชั่นอยู่แล้ว เราสามารถนำมาใช้ได้เลย โครงสร้างที่สำคัญได้แก่แอบสแตรกคลาสต่างๆและอินเตอร์เฟสที่ชื่อว่าคอลเลคชั่น

แอบสแตรกคลาส (Abstract Class)

แอบสแตรกคลาสคือคลาสที่ต้องมีอย่างน้อยหนึ่งเมธอดเป็นแอบสแตรกเมธอด แอบสแตรกเมธอดคือเมธอดที่ไม่มีโค้ดภายใน ทิ้งว่างไว้เพื่อให้ไปนิยามในคลาสลูกของมัน เราไม่สามารถสร้างออบเจ็กต์ขึ้นจากแอบสแตรกคลาสได้ (แต่สร้างจากลูกของมันที่นิยามโค้ดของแอบสแตรกเมธอดไว้ได้) หลักการจะคล้ายกับการใช้จาวาอินเตอร์เฟสนั่นเอง ตัวอย่างแอบสแตรกคลาสอยู่ในรูปที่ 4.1 อย่าลืมว่าต้องมีการบอกด้วยว่าคลาสไหนเมธอดไหนเป็นแอบสแตรก ด้วยคีย์เวิร์ด “abstract”

```
1:    public abstract class Robot{
2:        public String punch(){
3:            return “ Punch!”;
4:        }
5:
6:        public abstract String getClassName();
7:    }
```

รูป 4.1 แอบสแตรกคลาส

จุดมุ่งหมายในการใช้งานก็เหมือนกับอินเตอร์เฟส คือ เราสามารถใช้ไทป์ของข้อมูลร่วมกันได้ ตัวอย่างเช่นถ้าเราสร้างคลาสลูกของ Robot ออกมาสองคลาส ดังรูป 4.2 เราจะสามารถเอาออบเจ็กต์ทั้งสองชนิดไปใช้กับตัวแปรชนิดเดียวกันได้ดังรูป 4.3

```
1:    public class GetterRobot extends Robot{
2:        public abstract String getClassName(){
3:            return “GetterRobot”;
4:        }
5:    }
6:
7:    public class GundamRobot{
8:        public abstract String getClassName(){
9:            return “GundamRobot”;
10:       }
11:   }
```

รูป 4.2 คลาสลูกๆของ Robot

```
1:        ...
2:        Robot r;
3:        ...
4:        int robotSerialNumber = getSerialNumber();
5:        if (robotSerialNumber == 1)
6:            r = new GetterRobot();
7:        else
8:            r= new GundamRobot();
9:        System.out.println(r.getClassName()+r.punch());
10:       ...
```

รูป 4.3 การใช้งานแอบสแตรกคลาส

จากรูป 4.3 เมธอด getClassName ที่จะถูกเรียกใช้จะขึ้นอยู่กับตัวออบเจ็กต์ที่แท้จริง แต่เราให้ไทป์เป็น Robot ไว้ได้

ไทป์แบบมีพารามิเตอร์ (Parameterized Types, Generic Types)

จาวา 1.5 ขึ้นไปอนุญาตให้เราเขียนไทป์ของสมาชิกของคลาสได้ โดยเขียนในวงเล็บที่สร้างจากเครื่องหมาย “>” และ “<” สมมติว่าเราจะสร้างลิงค์ลิสต์ของจาวาเอง ซึ่งจะใช้เก็บข้อมูลประเภท Double (อย่าลืมว่า นี่ต้องเป็นออบเจ็กต์) เราสามารถเขียนโค้ดได้เป็น

LinkedList<Double> myList = new LinkedList<Double>();

ซึ่งเมื่อเขียนแบบนี้ไปแล้ว จะมีแต่ ข้อมูลประเภท Double เท่านั้นที่ลิสต์จะยอมรับ ดังนั้นในการใช้งานเมธอด get ที่เขียนไว้แล้วในลิงค์ลิสต์ (ยังไม่ต้องรู้มากเพราะมีเรื่องลิงค์ลิสต์ของจาวาต่างหากอีกที)

```
Double val = myList.get(o);
```

เราก็ไม่ต้องมาทำการเปลี่ยนรูปแบบข้อมูลที่ได้ให้เป็น Double เพราะว่าถูกบังคับไว้เรียบร้อยแล้ว

รูปแบบของไทป์ที่มีพารามิเตอร์ทั่วไป เป็นดังรูป 4.4 ซึ่งจะเห็นว่ามีพารามิเตอร์กี่ตัวก็ได้

```
class_or_interface_name<Class01, Class02, ...>
```

รูป 4.4 ไทป์ที่มีพารามิเตอร์ทั่วไป

ประโยชน์ของการเขียนไทป์แบบนี้คือ เราสามารถเห็นได้ทันทีว่าคอลเลคชั่นของเราเก็บอะไรอยู่ คอมไพเลอร์สามารถเช็คชนิดสมาชิกที่เราพยายามเอาใส่คอลเลคชั่นได้ทันที ถ้าไม่มีการกำหนดไทป์แบบนี้ก็จะคอมไพล์ผ่านแต่ไปเกิดข้อผิดพลาดตอนรัน

นอกจากนี้จาวา 1.5 ยังทำการเปลี่ยนชนิดระหว่างข้อมูลพื้นฐานกับ wrapper class ได้เองอีกด้วย นี่หมายความว่าเราสามารถเขียนโค้ดได้แบบข้างล่างนี้

- เราสามารถเขียน myList.add(3.8) แทน myList.add(new Double(3.8)) ได้เลย ตัวจาวาจะทำการ “boxing” ให้เป็นออบเจ็กต์เอง
- เราสามารถเขียน double sum = sum +myList.get(0); ได้ จาวาจะเปลี่ยนออบเจ็กต์ที่เอาออกมาจากลิสต์ให้เป็นตัวเลขโดยอัตโนมัติ เรียกว่าการ “unboxing”

คอลเลคชั่นอินเตอร์เฟส(Collection Interface)

ส่วนที่อยู่บนสุดของระดับคลาสในจาวาคอลเลคชั่นเฟรมเวิร์คก็คือ Collection และ Map ตัว Collection Interface นั้นมีเมธอดโดยสรุปดังรูปที่ 4.5 และมีการทำงานของแต่ละเมธอดดังรูปที่ 4.6

```
1:    public Interface Collection<E> extends Iterable<E>{
2:        boolean add(E o);
3:        boolean  addAll(Collection<? extends E> c);
4:        void clear();
5:        boolean contains(Object o);
6:        boolean containsAll(Collection<?> c);
7:        boolean equals(Object o);
8:        int hashCode();
9:        boolean isEmpty();
10:       Iterator<E> iterator();
11:       boolean remove(Object o);
12:       boolean removeAll(Collection<?> c);
13:       boolean retainAll(Collection<?> c);
14:       int size();
15:       Object[]toArray();
16:       <T> T[]toArray(T[] a);
17:   }
```

รูป 4.5 เมธอดของ Collection Interface

`boolean add(E o)` พยายามทำให้มี o อยู่ในคอลเลคชั่นนี้ รีเทิร์น true ถ้าการเรียกเมธอดนี้ทำให้ภายในคอลเลคชั่นเปลี่ยนไป รีเทิร์น false ถ้าคอลเลคชั่นนี้มีก็อปปี้ไม่ได้ และ มี o อยู่แล้ว
`boolean addAll(Collection<? extends E> c)` เติมสมาชิกจากคอลเลคชั่น c ทั้งหมดลงในคอลเลคชั่นของเรา แต่เมธอดนี้จะถือว่าใช้ไม่ได้ถ้า c ถูกเปลี่ยนแปลงระหว่างที่เมธอดนี้ทำงานอยู่ นั่นก็คือ c จะเป็นคอลเลคชั่นที่เราใช้เรียกเมธอดนี้เองไม่ได้ เมธอดนี้รีเทิร์น true ถ้าการเรียกเมธอดนี้ทำให้ภายในคอลเลคชั่นเปลี่ยนไป พารามิเตอร์ <? extends E> หมายถึงอะไรก็ได้ที่เป็นสับคลาสของ E โดย E คือพารามิเตอร์ของคอลเลคชั่นที่เรียกใช้เมธอดนี้ อย่าลืมว่าคลาสใดๆก็เป็นสับคลาสของตัวมันเอง
`void clear()` เอาสมาชิกของคอลเลคชั่นนี้ออกไปให้หมด

`boolean` **`contains`**`(Object o)` รีเทิร์น true ถ้าคอลเลคชั่นนี้มี o อยู่ หรือพูดได้อีกอย่างว่า รีเทิร์น true ก็ต่อเมื่อคอลเลคชั่นนี้มีสมาชิก e ซึ่ง (o==null ? e==null : o.equals(e)).
`boolean` **`containsAll`**`(Collection<?> c)` รีเทิร์น true ถ้าคอลเลคชั่นนี้มี สมาชิกจากคอลเลคชั่น c อยู่ทั้งหมด
`boolean` **`equals`**`(Object o)` เปรียบเทียบออบเจ็กต์ o กับคอลเลคชั่นนี้ว่าเท่ากันหรือไม่ โดยพื้นฐานแล้ว นี่คือเมธอดที่เอามาจากคลาส Object แต่ถ้าจะเขียนเมธอดนี้เอง เพื่อเปลี่ยนให้เป็นการเปรียบเทียบค่าที่อยู่ในคอลเลคชั่นก็ได้ โครงสร้างข้อมูล List กับ Set ของจาวานั้นได้เขียนเมธอดนี้ขึ้นเองโดยกำหนดให้ List ต้องเท่ากับ List และ Set ต้องเท่ากับ Set เท่านั้น ฉะนั้นถ้าเราเขียนเมธอดนี้ให้คลาสของเราเองซึ่งไม่ใช่ List หรือ Set แล้วละก็ เมธอดของเราจะรีเทิร์น false เมื่อนำไปใช้กับ List หรือ set นอกจากนี้ยังไม่สามารถสร้างคลาสซึ่ง implement List กับ Set ในเวลาเดียวกันได้
`int` **`hashCode`**`()` รีเทิร์นแฮชโค้ดของคอลเลคชั่นนี้ ถ้าเราโอเวอร์ไรด์ equals() แล้วเราต้องโอเวอร์ไรด์ hashcode() ด้วย เพราะ c1.equals(c2) ต้องหมายถึง c1.hashCode()==c2.hashCode()
`boolean` **`isEmpty`**`()` รีเทิร์น true ถ้าคอลเลคชั่นนี้ไม่มีสมาชิก
`Iterator<E>` **`iterator`**`()` รีเทิร์นอิเทอเรเตอร์ที่จะอนุญาตให้เราดูสมาชิกในคอลเลคชั่นนี้ได้ การเรียงของสมาชิกนั้นไม่ได้มีการกำหนดไว้ในชั้นนี้
`boolean` **`remove`**`(Object o)` เอาสมาชิก e ซึ่ง (o==null ? e==null : o.equals(e)) ออกจากคอลเลคชั่น ถ้ามีอยู่ในคอลเลคชั่น รีเทิร์น true ถ้ามีสมาชิกถูกเอาออกไปจริงๆ
`boolean` **`removeAll`**`(Collection<?> c)` เอาสมาชิกที่เหมือนกับสมาชิกใน c (โดยเทียบด้วย equals()) ทิ้ง รีเทิร์น true ถ้ามีสมาชิกถูกเอาออกไปจริงๆ c ห้ามเป็น null ไม่งั้นจะ throw exception
`boolean` **`retainAll`**`(Collection<?> c)` เอาสมาชิกที่เหมือนกับสมาชิกใน c (โดยเทียบด้วย equals()) เหลือไว้ นอกนั้น

ลบทิ้งหมด รีเทิร์น true ถ้ามีสมาชิกถูกเอาออกไปจริงๆ c ห้ามเป็น null ไม่งั้นจะ throw exception
`int size()` รีเทิร์นจำนวนสมาชิกในคอลเลคชั่นนี้ ถ้าจำนวนสมาชิกมากกว่า Integer.MAX_VALUE ให้รีเทิร์น Integer.MAX_VALUE.
`Object[] toArray()` รีเทิร์นอาร์เรย์ซึ่งใส่สมาชิกของคอลเลคชั่นนี้ไว้ทั้งหมด ถ้ามีการกำหนดลำดับของสมาชิกด้วยอิเทอเรเตอร์ไว้แล้ว ลำดับสมาชิกในอาร์เรย์ก็ต้องเป็นไปตามนั้นด้วย
`<T> T[] toArray(T[] a)` รีเทิร์นอาร์เรย์ซึ่งใส่สมาชิกของคอลเลคชั่นนี้ไว้ทั้งหมด รันไทม์ไทป์ของอาร์เรย์ที่รีเทิร์นให้เป็นชนิดเดียวกับอาร์เรย์ a ถ้าคอลเลคชั่นของเราใส่ a ได้ ก็จะเอาใส่ a แล้วรีเทิร์น a เลย (สมาชิกในด้านหลังของ a ที่มีที่เหลือก็จะถูกเซ็ตเป็น null) มิฉะนั้นต้องสร้างอาร์เรย์ใหม่ซึ่งขนาดใหญ่พอที่จะเก็บคอลเลคชั่นของเราได้ ถ้ามีการกำหนดลำดับของสมาชิกด้วยอิเทอเรเตอร์ไว้แล้ว ลำดับสมาชิกในอาร์เรย์ก็ต้องเป็นไปตามนั้นด้วย

รูป 4.6 การทำงานของเมธอดของคอลเลคชั่น

จะเห็นว่ามีชนิดข้อมูล E ซึ่งเวลาใช้งานจริงจะถูกแทนที่ด้วยข้อมูลชนิดอื่น คลาสที่อิมพลีเม้นท์คอลเลคชั่นอินเตอร์เฟสก็จะมีลักษณะดังนี้

```
public class LinkedList<E> implements Collection<E>. . .
```

ตอนใช้งาน สามารถแทน E ด้วยชนิดข้อมูลอื่น อย่างที่กล่าวไปแล้วข้างต้นในเรื่องของไทป์แบบมีพารามิเตอร์

อิเทอเรเตอร์(Iterator)

หน้าที่นั้นเหมือนกับในบทที่สาม คือมีหน้าที่ก่อให้เกิดการลูปไปยังสมาชิกต่างๆในคอลเลคชั่นได้ โดยผู้ใช้งานไม่ต้องเข้าหาฟิลด์ภายในของคอลเลคชั่นโดยตรง ทุกๆคลาสที่อิมพลีเม้นท์คอล

เลคชั่นจะต้องมีอิเทอเรเตอร์ ซึ่งจะต้องอิมพลีเม้นท์ตามอิเทอเรเตอร์อินเตอร์เฟสดังรูป 4.7 รายละเอียดของอิเทอเรเตอร์อินเตอร์เฟสอยู่ในรูปที่ 4.8

```
1:    public Interface Iterator<E>{
2:        boolean hasNext();
3:        E next();
4:        void remove();
5:    }
```

รูป 4.7 เมธอดของอิเทอเรเตอร์อินเตอร์เฟส

`boolean hasNext()` รีเทิร์น true ถ้ายังมีสมาชิกให้ดูได้อีก (นั่นคือ รีเทิร์น `true` เมื่อ `next()` จะรีเทิร์นสมาชิกนั่นเอง
`E next()` รีเทิร์นสมาชิกตัวต่อไปตามลำดับที่กำหนดไว้
`void remove()` เอาสมาชิกตัวที่พึ่งถูกรีเทิร์นด้วย next() ออกไป เรียกเมธอดนี้ได้หนึ่งครั้งต่อการเรียกใช้ next() หนึ่งครั้ง ถ้าตัวคอลเลคชั่นถูกเปลี่ยนระหว่างที่กำลังลูปด้วยวิธีอื่นที่ไม่ใช่เมธอดนี้ เราจะถือว่าเมธอดนี้ใช้ไม่ได้

รูป 4.8 คำอธิบายเมธอดของอิเทอเรเตอร์อินเตอร์เฟส

ตัวอย่างการใช้อิเทอเรเตอร์เป็นดังเช่นในรูป 4.9 ซึ่งเราได้สร้างอิเทอเรเตอร์ของคอลเลคชั่นของเรา (myCollection) ขึ้นมา โดย myCollection นั้นเก็บข้อมูลประเภท Integer จากนั้นเราลูปแล้วเลือกพิมพ์เฉพาะตัวเลขที่มีค่ามากกว่าหนึ่งร้อย

เราต้องระวังการลูปที่ผิด อย่าลืมว่าการลูปแต่ละครั้งควรใช้ next() แค่ครั้งเดียวเท่านั้น มิฉะนั้นเราจะทำเกินตำแหน่งที่ต้องการ รูป 4.10 แสดงตัวอย่างที่ตั้งใจจะเขียนแบบรูป 4.9 แต่ทำการลูปผิดวิธี การใช้ next() สองทีทำให้พิมพ์จำนวนลำดับที่ถัดจากเลขที่ต้องการออกมา

```
1:    ...
2:        Iterator<Integer> itr = myCollection.iterator();
3:        int currentNumber;
4:        while(itr.hasNext()){
5:            currentNumber = itr.next(); //auto convert
6:            if(currentNumber>100)
7:                System.out.println(currentNumber);
8:        }
9:    ...
```

รูป 4.9 การลูปเลือกพิมพ์บางสมาชิกของคอลเลคชั่น

```
1:    ...
2:        Iterator<Integer> itr = myCollection.iterator();
3:        int currentNumber;
4:        while(itr.hasNext()){
5:            if(itr.next() >100)
6:                System.out.println(itr.next());
7:        }
8:    ...
```

รูป 4.10 การลูปที่ผิดวิธี เพราะใช้ next() หลายครั้ง

การลูปโดยใช้ อิเทอเรเตอร์นั้น เราอาจใช้ for loop แบบพิเศษได้ รูป 4.11 แสดงโค้ดในรูปที่ 4.9 หลังจากเปลี่ยนไปใช้ for loop แบบพิเศษ ซึ่งเครื่องหมาย ":" นั้นหมายถึง "ใน" นั่นก็หมายความว่า โค้ดของส่วน for loop ในรูปที่ 4.11 จะอ่านได้ว่า "สำหรับแต่ละจำนวนเต็มใน myCollection"

```
1:    ...
2:        Iterator<Integer> itr = myCollection.iterator();
3:        for(Integer currentInt : myCollection ){
4:          if(currentInt.intValue() >100)
5:             System.out.println(currentInt.intValue());
6:        }
7:    ...
```

รูป 4.11 for loop แบบพิเศษ

แต่ว่า for loop แบบพิเศษนั้นไม่สามารถใช้ได้ในกรณีที่ตัวคอลเลคชั่นต้องมีการเปลี่ยนแปลงระหว่างการลูป ดังนั้นการลูปเอาสมาชิกออกจากคอลเลคชั่นก็ต้องใช้อิเทอเรเตอร์เท่านั้น รูป 4.12 แสดงการเอาสมาชิกออกจากคอลเลคชั่น ในตัวอย่างนี้จะใช้ for loop แบบธรรมดา

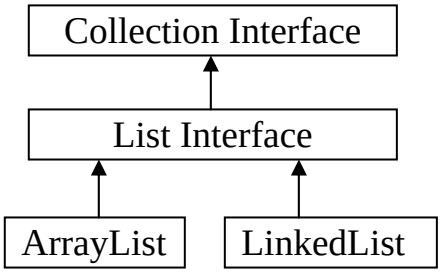
```
1:    ...
2:      int currentNumber;
3:      for(Iterator<Integer> itr =
4:      myCollection.iterator(); itr.hasNext(); ){
5:          currentNumber = itr.next(); //auto convert
6:          if(currentNumber<100)
7:              itr.remove();
8:      }
9:    ...
```

รูป 4.12 การลูปเอาสมาชิกออกจากคอลเลคชั่น

ลิสต์อินเตอร์เฟส(List Interface)

ลิสต์ของจาวาคือที่เก็บของเรียงกัน โดยสามารถเข้าถึงของแต่ละชิ้นโดยการใช้ดัชนี (index) ได้ ดัชนีนั้นมีค่าเริ่มจากศูนย์ และภายในลิสต์อนุญาตให้มีของซ้ำกันได้ สิ่งที่เป็นลิสต์มักมี null เป็นสมาชิกได้ ลิสต์อินเตอร์เฟสมีอิเทอเรเตอร์ของมันเองเรียกว่า ลิสต์อิเทอเรเตอร์ (ListIterator) ซึ่งมีเมธอดสำหรับการใส่ของและเอาของออกจากลิสต์ รวมทั้งสามารถอนุญาตการลูปได้สองทิศทางอีกด้วย ลิสต์เป็นอินเตอร์เฟส ดังนั้นโครงสร้างภายในอาจสร้างขึ้นมาจากอะไรก็ได้ จาวามีคลาสแอ็บสแตรกลิสต์ (AbstractList) เป็นสับคลาสของลิสต์ซึ่ง แอ็บสแตรกลิสต์ อิมพลีเม้นท์โค้ดของลิสต์บางส่วน นอกนั้นจะเป็นหน้าที่ของสับคลาสของแอ็บสแตรกลิสต์อีกทีหนึ่ง ซึ่งจริงๆแล้วมีสองคลาสคือ อาร์เรย์ลิสต์ (ArrayList) กับลิงค์ลิสต์(LinkedList) (ต่อไปจะขอใช้ตัวภาษาอังกฤษสำหรับคลาสเหล่านี้เพื่อไม่ให้เกิดความสับสน) ArrayList นั้นสร้างลิสต์จากอาร์เรย์ ส่วน LinkedList นั้นสร้างจากลิงค์ลิสต์แบบสองทาง รูป 4.13 แสดงโครงสร้างลำดับคลาสของอินเตอร์เฟสนี้

เมื่อสร้างตัวแปรให้เป็นชนิด List เราก็สามารถเปลี่ยนไปใช้ ArrayList หรือ LinkedList เมื่อใดก็ได้ โครงสร้างอินเตอร์เฟสของ List อยู่ในรูป 4.14 ส่วนรายละเอียดอยู่ในรูป 4.15



รูป 4.13 โครงสร้างลำดับคลาสของลิสต์อินเตอร์เฟส

```
1:    public interface List<E> extends Collection<E>{
2:       boolean add(E o);
3:       void add(int index, E element);
4:       boolean addAll(Collection<? extends E> c);
5:       boolean addAll(int index, Collection<? extends
E> c);
6:       void clear();
7:       boolean contains(Object o);
8:       boolean containsAll(Collection<?> c);
9:       boolean equals(Object o);
10:      E get(int index);
11:      int hashCode();
12:      int indexOf(Object o);
13:      boolean isEmpty();
14:      Iterator<E> iterator();
15:      int lastIndexOf(Object o);
16:      ListIterator<E> listIterator();
17:      ListIterator<E> listIterator(int index);
18:      E remove(int index);
19:      boolean remove(Object o);
20:      boolean removeAll(Collection<?> c);
21:      boolean retainAll(Collection<?> c);
22:    E set(int index, E element);
23:      int size();
24:      List<E> subList(int fromIndex, int toIndex);
25:      Object[] toArray();
26:      <T> T[] toArray(T[] a);
27:   }
```

รูป 4.14 List Interface

`boolean` **`add`**`(E o)` ใส่ o ไปที่ท้ายลิสต์ ลิสต์ที่เราสร้างขึ้นอาจกำหนดชนิดของข้อมูลที่ใส่ได้เอาไว้ เช่น ไม่รับค่า null หรือไม่รับข้อมูลบางชนิด เอกสารของลิสต์แต่ละชนิดควรบอกถึงชนิดของข้อมูลที่ไม่รับด้วย รีเทิร์น true ถ้าการเรียกเมธอดนี้ทำให้ภายในคอลเลคชั่นเปลี่ยนไป
`void` **`add`**`(int index, E element)` ใส่ element เข้าไปที่ตำแหน่งที่ index เลื่อนสมาชิกในลิสต์ตัวที่เคยอยู่ตรงนั้นรวมทั้งสมาชิกตัวถัดไปอื่นๆ ไปทางขวาตัวละตำแหน่ง
`boolean` **`addAll`**`(Collection<? extends E> c)` ใส่ของใน c ต่อท้ายลิสต์ ลำดับที่ใส่นั้นเป็นไปตามอิเทอเรเตอร์ของ c เมธอดนี้จะใช้ไม่ได้ถ้า c ถูกเปลี่ยนแปลงในระหว่างที่เรียกใช้เมธอดนี้ เช่นเมื่อ c คือตัวลิสต์เอง รีเทิร์น true ถ้ามีการเติมสมาชิกลงไปจริง
`boolean` **`addAll`**`(int index, Collection<? extends E> c)` ใส่ของใน c ลงไปในลิสต์ ณ ตำแหน่งที่ถูกกำหนดโดย ลำดับที่ใส่นั้นเป็นไปตามอิเทอเรเตอร์ของ c เลื่อนสมาชิกในลิสต์ตัวที่เคยอยู่ตำแหน่งนั้น รวมทั้งสมาชิกตัวถัดไปอื่นๆ ไปทางขวา เมธอดนี้จะใช้ไม่ได้ถ้า c ถูกเปลี่ยนแปลงในระหว่างที่เรียกใช้
`void` **`clear`**`()` ทำให้ลิสต์นี้ว่าง
`boolean` **`contains`**`(Object o)` รีเทิร์น true ถ้าลิสต์นี้มี o อยู่ หรือพูดได้อีกอย่างว่า รีเทิร์น true ก็ต่อเมื่อคอลเลคชั่นนี้มีสมาชิก e ซึ่ง (o==null ? e==null : o.equals(e))
`boolean` **`containsAll`**`(Collection<?> c)` รีเทิร์น true ถ้าลิสต์นี้มี สมาชิกจากคอลเลคชั่น c อยู่ทั้งหมด
`boolean` **`equals`**`(Object o)` เปรียบเทียบออบเจ็กต์ o กับลิสต์นี้ว่าเท่ากันหรือไม่ รีเทิร์น true ถ้า o ก็เป็นลิสต์ ลิสต์ของเรากับ o มีขนาดเท่ากัน และมีสมาชิกเหมือนกัน (เทียบด้วยเมธอด equals()) เรียงกันด้วยลำดับเดียวกันทุกประการ
`E` **`get`**`(int index)` รีเทิร์นสมาชิก ณ ตำแหน่งที่บอกด้วย index
`int` **`hashCode`**`()` รีเทิร์นแฮชโค้ดของลิสต์นี้ โดยมีสูตรว่า

```
hashCode = 1;
Iterator i = list.iterator();
while (i.hasNext()){
    Object obj = i.next();
    hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
}
```

`int` **`indexOf`**`(Object o)`

รีเทิร์นตำแหน่งที่มี o อยู่เป็นตำแหน่งแรก หรือ -1 ถ้า o ไม่ได้อยู่ในลิสต์นี้

`boolean` **`isEmpty`**`()`

รีเทิร์น true ถ้าลิสต์นี้เป็นลิสต์ว่าง

`Iterator<E>` **`iterator`**`()`

รีเทิร์นอิเทอเรเตอร์

`int` **`lastIndexOf`**`(Object o)`

รีเทิร์นตำแหน่งที่มี o อยู่เป็นตำแหน่งสุดท้าย หรือ -1 ถ้า o ไม่ได้อยู่ในลิสต์นี้

`ListIterator<E>` **`listIterator`**`()`

รีเทิร์นลิสต์อิเทอเรเตอร์

`ListIterator<E>` **`listIterator`**`(int index)`

รีเทิร์นลิสต์อิเทอเรเตอร์ โดยให้ตำแหน่งที่สนใจเริ่มจากตำแหน่ง index นี่จะเป็นตำแหน่งที่เมธอด next() รีเทิร์นมาเป็นตำแหน่งแรก ส่วนการเรียก previous() ครั้งแรกจากสถานะนี้จะรีเทิร์นสมาชิกตัวที่มีตำแหน่งน้อยกว่านี้หนึ่งตำแหน่ง

`E` **`remove`**`(int index)`

เอาสมาชิกที่ตำแหน่ง index ออกจากลิสต์ รีเทิร์นสมาชิกนั้นออกมา ส่วนสมาชิกตัวอื่นก็เลื่อนมาแทนที่ตำแหน่งของตัวที่ออกไปนี้

`boolean` **`remove`**`(Object o)`

เอาสมาชิกที่ค่าเท่ากับ o (เทียบด้วย equals()) อยู่เป็นตำแหน่งแรกออกจากลิสต์ไป ถ้า o ไม่ได้อยู่ในลิสต์นี้ก็จะไม่มีอะไรเปลี่ยนแปลง รีเทิร์น true ถ้ามีสมาชิกที่มีค่าเท่ากับ o อยู่ในลิสต์จริงๆ

`boolean` **`removeAll`**`(Collection<?> c)`

เอาของที่อยู่ใน c ออกจากลิสต์ไปให้หมด รีเทิร์น true ถ้าลิสต์เกิดการเปลี่ยนแปลง

`boolean` **`retainAll`**`(Collection<?> c)`

เอาสมาชิกที่เหมือนกับสมาชิกใน c (โดยเทียบด้วย equals()) เหลือไว้ นอกนั้น

ลบทิ้งหมด รีเทิร์น true ถ้ามีสมาชิกถูกเอาออกไปจริงๆ c ห้ามเป็น null ไม่งั้นจะ throw exception
`E set(int index, E element)` เปลี่ยนสมาชิก ณ ตำแหน่ง index ให้เป็น element รีเทิร์นสมาชิกตัวที่อยู่ก่อนจะถูกเปลี่ยน
`int size()` รีเทิร์นจำนวนสมาชิกในคอลเลคชั่นนี้ ถ้าจำนวนสมาชิกมากกว่า Integer.MAX_VALUE ให้รีเทิร์น Integer.MAX_VALUE
`List<E> subList(int fromIndex, int toIndex)` รีเทิร์นลิสต์ย่อย นับรวมตั้งแต่ตำแหน่ง fromIndex จนถึงตำแหน่ง toIndex-1 (ถ้า fromIndex เท่ากับ toIndex เมธอดนี้จะรีเทิร์นลิสต์ว่าง) การเปลี่ยนแปลงในลิสต์ย่อยจะเห็นผลที่ลิสต์ใหญ่ด้วย และการเปลี่ยนแปลงที่ลิสต์ใหญ่ก็จะเห็นผลที่ลิสต์ย่อย ลิสต์ย่อยที่รีเทิร์นมาสามารถใช้เมธอดได้แบบลิสต์ใหญ่ตัวที่เรียกใช้เมธอดนี้ เราสามารถใช้ลิสต์ย่อยในการทำงานเฉพาะส่วนได้เช่น list.subList(from, to).clear(); ถ้าลิสต์ตัวใหญ่ถูกเปลี่ยนขนาดหรือถูกทำให้ใช้งานอิเทอเรเตอร์ไม่ได้ผล เมธอดนี้ก็จะใช้ไม่ได้
`Object[] toArray()` เหมือนกับเมธอดของคอลเลคชั่น
`<T> T[] toArray(T[] a)` เหมือนกับเมธอดของคอลเลคชั่น

รูป 4.15 รายละเอียดของ List Interface

ลิสต์อิเทอเรเตอร์(List Iterator)

ลิสต์อิเทอเรเตอร์นั้น ต่างจากอิเทอเรเตอร์ที่นิยามไว้ในคอลเลคชั่นธรรมดาเพราะว่าลิสต์อิเทอเรเตอร์นั้นสามารถเคลื่อนตำแหน่งได้สองทิศ สมาชิกปัจจุบันที่ลิสต์อิเทอเรเตอร์สนใจนั้นไม่มี จุดที่สนใจคือจุดที่อยู่ระหว่างสมาชิกที่จะถูกรีเทิร์นจาก next() กับสมาชิกที่จะถูกรีเทิร์นจาก previous() ลิสต์อิเทอเรเตอร์นั้นเป็นสับคลาสของอิเทอเรเตอร์

public interface **ListIterator<E>** extends Iterator<E>

`void add(E o)` ใส่ o ลงไปในลิสต์
`boolean hasNext()` รีเทิร์น true ถ้ายังมีสมาชิกของลิสต์ให้ดูอยู่ ในทิศทางที่ไปข้างหน้า (นั่นคือ ถ้าการเรียก next() ครั้งต่อไปรีเทิร์นสมาชิกในลิสต์มา)
`boolean hasPrevious()` รีเทิร์น true ถ้ายังมีสมาชิกให้ดูในทิศทางย้อนกลับ (นั่นคือ ถ้าการเรียก previous() ครั้งต่อไปรีเทิร์นสมาชิกในลิสต์มา)
`E next()` รีเทิร์นสมาชิกตัวถัดไปในลิสต์ ถ้าเราเรียก next() แล้ว เรียก previous() สลับกันครั้งต่อครั้ง จะได้สมาชิกตัวเดียวกันเป็นผลลัพธ์ตลอด
`int nextIndex()` รีเทิร์นค่าดัชนีของสมาชิกตัวที่จะเป็นคำตอบของ next() หรือถ้าอยู่ที่ตำแหน่งสุดท้ายของลิสต์แล้วก็ให้รีเทิร์นขนาดของลิสต์
`E previous()` รีเทิร์นสมาชิกตัวก่อนในลิสต์
`int previousIndex()` รีเทิร์นค่าดัชนีของสมาชิกตัวที่จะเป็นคำตอบของ previous() หรือถ้าอยู่ที่ตำแหน่งแรกของลิสต์แล้วก็ให้รีเทิร์น-1
`void remove()` เอาสมาชิกตัวที่พึ่งเป็นผลลัพธ์ของ next() หรือ previous() ออกจากลิสต์ เมธอดนี้ใช้สลับกับ next() หรือ previous()ได้ครั้งต่อครั้งเท่านั้น จะต้องไม่มีการเรียกเมธอด ListIterator.add() ระหว่างการเรียก next() หรือ previous() กับเมธอดนี้
`void set(E o)` เอา o แทนที่สมาชิกตัวท้ายสุดที่พึ่งถูกรีเทิร์นเป็นคำตอบของ next() หรือ previous() นอกจากนี้จะต้องไม่มีการเรียกเมธอด ListIterator.add() และ ListIterator.remove() ระหว่างการเรียก next() หรือ previous() กับเมธอดนี้

รูป4.16 เมธอดของลิสต์อิเทอเรเตอร์

รูป 4.17 แสดงตัวอย่างการลูปถอยหลังดูสมาชิกในลิสต์ เมื่อเรามีลิสต์ e ส่วนรูปที่ 4.18 เป็นการลูปจากตอนต้นของลิสต์ eไป โดยแทนที่ทุกออบเจ็กต์ที่มีค่าเท่ากับ val ด้วย new

```
1:      ...
2:       ListIterator<Double> i;
3:        for(i= e.listIterator(e.size());i.hasPrevious(); )
4:           System.out.println(i.previous());
5:      ...
```

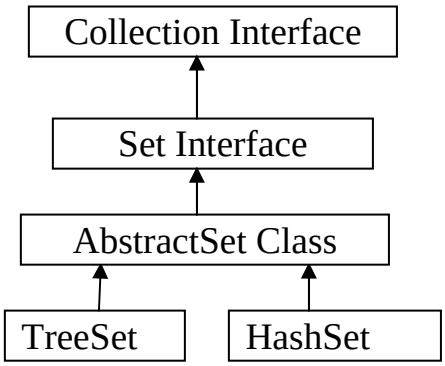
รูป 4.17 การลูปถอยหลังในจาวาลิสต์โดยใช้อิเทอเรเตอร์ โดยเริ่มจากท้ายลิสต์

```
1:      ...
2:        ListIterator<Double> i;
3:        for(i= e.listIterator();i.hasNext(); )
4:           if(val.equals(i.next()))
5:              i.set(new);
6:      ...
```

รูป 4.18 การลูปเดินหน้าแทนที่ val ด้วย new

เซตอินเตอร์เฟส(Set Interface)

เซตนั้นคือคอลเลคชั่นที่ห้ามมีสมาชิกซ้ำ เมธอดของเซตนั้นมีจำนวนเท่ากับเมธอดของคอลเลคชั่น โค้ดของเซตนั้นสร้างจากคลาสแอ็บสแตรกเซต (AbstractSet) ส่วนหนึ่ง นอกนั้นจะเป็นหน้าที่ของสับคลาสของแอ็บสแตรกเซตอีกทีหนึ่ง ซึ่งจริงๆแล้วมีสองคลาสคือ ทรีเซต (TreeSet) กับ แฮชเซต (HashSet) โดยมีโครงสร้างดังรูป 4.19



รูป 4.19 โครงสร้างของ Set

การใช้งานก็เพียงแต่รู้จักสร้างเซตให้เป็น แล้วเรียกเมธอดให้ถูกต้อง อย่าลืมว่าเซตจะมีสมาชิกซ้ำกันสองตัวหรือมากกว่านั้นไม่ได้ ถ้าหากเรามีคอลเลคชั่น c อยู่แล้วต้องการทำให้ไม่มีสมาชิกที่ซ้ำกัน เราสามารถสร้างเซตขึ้นจาก c ได้โดยเลือกใช้คอนสตรัคเตอร์ที่รับ c เป็นพารามิเตอร์ เช่น Collection<Double> d = new HashSet(c);

ทรีเซตนั้นภายในสร้างขึ้นมาด้วยโครงสร้างต้นไม้ ลำดับในการไล่ดูสมาชิกด้วยอิเทอเรเตอร์จะเรียงเป็นรูปแบบที่แน่นอน(ดังนั้นสมาชิกภายในทรีเซตต้องเป็น Comparable) ส่วนแฮชเซตนั้นสร้างขึ้นมาด้วยโครงสร้างข้อมูลตารางแฮช ซึ่งจะมีการเรียงของสมาชิกภายในค่อนข้างสุ่ม (ตอนนี้ยังไม่ต้องสนใจว่า HashSet กับ TreeSet ต่างกันอย่างไร เราใช้ได้ทั้งคู่) เซตนั้นใช้เมธอดชื่อเหมือนกับคอลเลคชั่นทุกเมธอด เมธอดที่ทำงานต่างจากในคอลเลคชั่นอยู่ในรูปที่ 4.20

`boolean` **add**(E o) เติมสมาชิกใหม่ลงไปในเซตถ้ามันไม่ซ้ำกับสมาชิกที่มีอยู่แล้ว นั่นคือ เติม o ลงไปในเซตถ้าไม่มี e ซึ่ง (o==null ? e==null : o.equals(e)) ถ้ามีสมาชิกตัวที่เราต้องการเติมอยู่ในเซตแล้ว เซตจะไม่เปลี่ยนแปลงและเมธอดนี้จะรีเทิร์น false.
`boolean` **addAll**(Collection<? extends E> c) เหมือนกับเมธอดของ Collection Interface เพียงแต่จะไม่เติมสมาชิกที่ซ้ำ นั่นคือถ้า c เป็นเซต เราก็จะได้การยูเนี่ยนของสองเซตนั่นเอง
`boolean` **equals**(Object o) รีเทิร์น true ถ้า o เป็นเซตที่มีขนาดเดียวกับเซตที่เรียกใช้เมธอดนี้และทุกสมาชิกที่อยู่ใน o อยู่ในเซตที่เรียกใช้เมธอดนี้ อย่าลืมว่า o ต้องเป็น Set ด้วย
`int` **hashCode**() รีเทิร์นค่าแฮชโค้ด ซึ่งแฮชโค้ดของเซต คือผลบวกของแฮชโค้ดจากสมาชิกในเซต (แฮชโค้ดของ null คือ 0)
Iterator<E> **iterator**() รีเทิร์นอิเทอเรเตอร์ การเรียงของสมาชิกจะขึ้นอยู่กับว่าจริงๆแล้วเซตนี้เป็นแฮชเซตหรือทรีเซต

รูป 4.20 เมธอดของเซตที่ต่างจากคอลเลคชั่นธรรมดา

ตัวอย่างที่ 4-1

จงเขียนโปรแกรมซึ่งรับสตริงเข้ามาจากเมนแล้วพิมพ์คำที่ซ้ำในสตริงนั้นออกมาทุกคำ นอกจากนี้จงพิมพ์คำที่ใช้ในสตริงนี้ออกมาโดยไม่ให้มีคำซ้ำกัน

เราสามารถทำโจทย์ข้อนี้ได้โดยใช้เซต โดยมีโค้ดดังรูป 4.21

```
1:    import java.util.*;
2:    public class FindDups {
3:       public static void main(String args[]) {
4:          Set<String> s = new HashSet();
5:          for (int i=0; i<args.length; i++){
6:             if (!s.add(args[i])) //เติมสมาชิก  ถ้าไม่สำเร็จจะรีเทิร์น
7:                                  //false
8:                System.out.println("คำซ้ำ: "+args[i]);
9:          }
10:         System.out.println("คำที่ใช้มี: "+ s);
11:      }
12:   }
```

รูป 4.21 โปรแกรมหาคำซ้ำและคำที่ใช้ในประโยค

ในการใช้เมธอดเช่น addAll(c) หรือ removeAll(c) นั้น ตัวเซตต้นฉบับของเราจะเกิดการเปลี่ยนแปลง ถ้าเราไม่ต้องการให้เกิดการเปลี่ยนแปลง ก็ต้องสร้างตัวก็อปปี้ขึ้นมาดังตัวอย่างต่อไปนี้

ตัวอย่างที่ 4-2

จงเขียนเมธอดของโปรแกรมซึ่งรับเซตสองเซตเป็นอินพุตแล้วรีเทิร์นเซตที่เกิดจากการอินเตอร์เซกสองเซตที่เป็นอินพุตเข้าด้วยกัน

ข้อควรระวังก็คือ เราไม่ควรไปเปลี่ยนเซตที่ได้มาเป็นอินพุตโดยไม่จำเป็น ดังนั้นจึงต้องสร้างโค้ดให้มีตัวก็อปปี้ของเซตไว้ดังรูปที่ 4.22

```
1:    ...
2:       public static Set union(Set s1, Set s2) {
3:          Set intersect = new HashSet(s1);
4:          intersect.retainAll(s2);
5:          return intersect;
6:       }
7:    ...
```

รูป 4.22 โปรแกรมอินเตอร์เซกเซตสองเซต

คราวนี้มาลองดูอีกหนึ่งตัวอย่าง

ตัวอย่างที่ 4-3

จงเขียนโปรแกรมซึ่งรับสตริงเข้ามาจากแมนแล้วสร้างเซตสองเซต เซตหนึ่งเก็บคำที่ไม่มีการซ้ำเกิดขึ้นเลย ส่วนอีกเซตหนึ่งเก็บคำที่มีการซ้ำ พิมพ์เซตสองเซตนี้ออกที่หน้าจอด้วย

เราสามารถเขียนโค้ดได้ดังรูป 4.23

```
1:    import java.util.*;
2:    public class FindDups2 {
3:       public static void main(String args[]) {
4:          Set uniques = new HashSet();
5:          Set dups = new HashSet();
6:          for (int i=0; i<args.length; i++){
7:             if (!uniques.add(args[i]))
8:                dups.add(args[i]);
9:          }
10:         uniques.removeAll(dups);
11:         System.out.println("Unique words: " + uniques);
12:         System.out.println("Duplicate words: " + dups);
13:      }
14:   }
```

รูป 4.23 โปรแกรมพิมพ์เซตของข้อความที่ไม่ซ้ำและซ้ำ

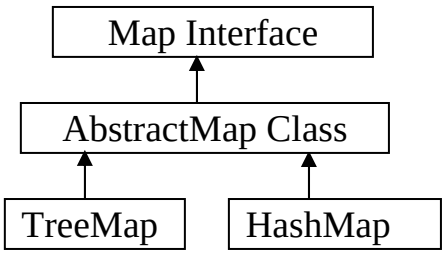
แม็ปอินเตอร์เฟส(Map Interface)

เป็นอินเตอร์เฟสที่สมาชิกแต่ละตัวเป็นคู่ (key, value) โดยค่าของคีย์จะต้องไม่ซ้ำกับคีย์อื่นๆใน แม็ปเดียวกัน อินเตอร์เฟสนี้ใช้เก็บข้อมูลที่มีความสัมพันธ์กันนั่นเอง

แม็ปนั้นไม่ได้เป็นสับอินเตอร์เฟสของคอลเลคชั่น โครงสร้างระดับคลาสของแม็ปอยู่ในรูปที่ 4.24 โดยมีส่วนหนึ่งของโค้ดอยู่ในคลาสแอ็บสแตรคแม็ป (AbstractMap) และนอกนั้นอยู่ใน คลาสทรีแม็ป(TreeMap) กับ แฮชแม็ป(HashMap)

เราสามารถสร้างแม็ปใหม่โดยเอาสมาชิกมาจากแม็ปที่มีอยู่แล้วได้ คอนสตรัคเตอร์ของคลาสทรี แม็ปกับแฮชแม็ปนั้นได้ถูกเขียนเตรียมไว้เพื่อการนี้แล้ว ถ้าเรามีแม็ป m อยู่ เราเอาไปสร้างแม็ป ใหม่ได้ง่ายๆดังนี้

```
Map newMap = new HashMap(m);
```



รูป 4.24 โครงสร้างระดับคลาสของแม็ป

แม็ปเอ็นทรี่(Map Entry)

คู่ (key, value) นั้นถูกเก็บไว้ในโครงสร้างข้อมูลแม็ปเอ็นทรี่ (**Map.Entry<K, V>**) เมธอดของ แม็ปเอ็นทรี่นั้นเป็นดังแสดงในรูป 4.25 ตัวแม็ปเอ็นทรี่นั้นเป็นออบเจ็กต์ที่เป็นสมาชิกของเซตที่ รีเทิร์นมาจากเมธอด Map.entrySet() ทางเดียวที่จะเข้าถึงแม็ปเอ็นทรี่ได้คือผ่านทางอิเทอเรเตอร์

ของเซตนี้เท่านั้น ตัวแม็ปเอ็นทรี่นั้นมีเมธอดเดียวที่ใช้เปลี่ยนค่าภายใน คือ setValue() ส่วนเมธอดของตัวแม็ปเองนั้นอยู่ในรูปที่ 4.26

`boolean` **equals**(Object o)
เทียบ o กับแม็ปเอ็นทรี่นี้ว่าเป็นแม็ปเอ็นทรี่เดียวกันหรือไม่ โดยมีนิยามการเทียบว่า e1 และ e2 ถือเป็นแม็ปเอ็นทรี่ที่ค่าเท่ากันเมื่อ
`(e1.getKey()==null ?`
`e2.getKey()==null : e1.getKey().equals(e2.getKey())) &&`
`(e1.getValue()==null ?`
`e2.getValue()==null : e1.getValue().equals(e2.getValue()))`

K **getKey**() รีเทิร์นคีย์ของแม็ปเอ็นทรี่นี้
V **getValue**() รีเทิร์นตัวค่าของแม็ปเอ็นทรี่นี้ ถ้า iterator.remove() ได้ลบค่านี้ออกไปจากแม็ปแล้ว เมธอดนี้ก็ถือว่าใช้การไม่ได้

`int` **hashCode**()
แฮชโค้ดของแม็ปเอ็นทรีใช้สูตรดังนี้
`(e.getKey()==null ? 0 : e.getKey().hashCode()) ^`
`(e.getValue()==null ? 0 : e.getValue().hashCode())`

V **setValue**(V value) เปลี่ยนตัวค่าของแม็ปเอ็นทรี่นี้ให้เป็นค่าใหม่ รีเทิร์นค่าเก่าออกมา นี้ ถ้า iterator.remove() ได้ลบค่านี้ออกไปจากแม็ปแล้ว เมธอดนี้ก็ถือว่าใช้การไม่ได้

รูป 4.25 เมธอดของ Map.Entry

การใช้งานเมธอดของแม็ป

แม็ปนั้นมีหลายเมธอดให้ใช้ดังรูป 4.26

`void`	**clear**() เอาของออกไปให้หมด
`boolean`	**containsKey**(Object key) รีเทิร์น true ถ้าแม็ปนี้มีค่าคีย์ k ซึ่งเมื่อเทียบด้วย equals() แล้วมีค่าเท่ากับ key

boolean	**containsValue**(Object value) รีเทิร์น true ถ้าแม็ปนี้มีคีย์ ซึ่งแม็ปกับ value
Set<Map.Entry<K, V>>	**entrySet**() รีเทิร์นคู่ (key,value) ทั้งหมดของแม็ปนี้ ออกมาเป็นเซตของแม็ปเอ็นทรี่ เซตนี้กับตัวแม็ปนั้นมีการเปลี่ยนแปลงไปด้วยกัน ห้ามเปลี่ยนแม็ปนี้เด็ดขาดระหว่างที่เราทำการอิเทอเรทบนเซตนี้ (ยกเว้นว่าเปลี่ยนด้วย iterator.remove() ของเซตนี้เอง หรือด้วย setValue() ของแม็ปเอ็นทรี่ที่รีเทิร์นมาจากอิเทอเรเตอร์) เซตนี้ใช้เมธอดได้คือ Iterator.remove(), Set.remove(), removeAll(), retainAll() และ clear() ส่วนเมธอด add() กับ addAll() นั้นไม่มีใช้
boolean	**equals**(Object o) ทดสอบว่าแม็ปนี้เท่ากับ o หรือไม่ แม็ปสองแม็ป (t1 และ t2) จะถือว่าเท่ากันถ้า t1.entrySet().equals(t2.entrySet()).
V	**get**(Object key) รีเทิร์น value ที่แมทช์กับ key ในแม็ปนี้ รีเทิร์น null ถ้าในแม็ปไม่มีคีย์ key แต่ว่าการรีเทิร์น null ไม่ได้หมายความว่าไม่มีคีย์ key เพราะตัวที่ key แม็ปไปอาจมีค่าเป็น null ก็ได้ ดังนั้นต้องใช้เมธอด containsKey() ตรวจสอบดูอีกที
int	**hashCode**() รีเทิร์นแฮชโค้ดของแม็ปนี้ แฮชโค้ดของแม็ปคือผลบวกของแฮชโค้ดของแต่ละเอ็นทรี่ใน entrySet ของแม็ปนี้นั่นเอง
boolean	**isEmpty**() รีเทิร์น true ถ้าแม็ปนี้ไม่มีอะไรข้างใน
Set<K>	**keySet**()

	รีเทิร์นเซตของคีย์ในแม็ปนี้ ถ้ามีการเปลี่ยนเซตนี้ แม็ปก็จะเปลี่ยนตามด้วย ถ้าเปลี่ยนแม็ปเซตนี้ก็จะเปลี่ยนตามเช่นเดียวกัน ในระหว่างที่เราทำการลูปบนเซตที่ได้มานี้ ห้ามมีการเปลี่ยนอะไรบนแม็ปนี้เด็ดขาดนอกจากใช้ iterator.remove() ของเซตนี้เอง เซตนี้ใช้เมธอดได้คือ Iterator.remove(), Set.remove(), removeAll(), retainAll() และ clear() ส่วนเมธอด add() กับ addAll() นั้นไม่มีใช้
V	**put**(K key, V value) เอาคู่ key, value ใส่แม็ป ถ้าแม็ปนี้เคยมีคีย์ key อยู่แล้ว ให้เขียน value ทับค่าเก่าซะ รีเทิร์นค่าเก่าที่แม็ปกับ key หรือรีเทิร์น null ถ้า key นี้พึ่งเข้าสู่แม็ปนี้เป็นครั้งแรก แต่อย่าลืมว่า null อาจหมายถึงว่าค่าที่คีย์ key นั้นแม็ปไว้คือ null
void	**putAll**(Map<? extends K,? extends V> t) เอาของในแม็ป t ใส่แม็ปของเรา เมธอดนี้ก็คือการทำเมธอด put หลายๆครั้งนั่นเอง โดยแต่ละครั้งก็ใช้สมาชิกของ t หนึ่งตัว
V	**remove**(Object key) เอาทั้งคีย์ที่ตรงกับ key และค่าของมันออกจากแม็ป
int	**size**() รีเทิร์นจำนวนคู่ของ key-value ในแม็ปนี้มา
Collection<V>	**values**() รีเทิร์นคอลเลคชั่นวิวของ values ในแม็ปนี้ออกมา

รูป 4.26 เมธอดของแม็ป

ตัวอย่างที่ 4-4

จงสร้างโปรแกรมที่รับอินพุตเป็นคำต่างๆแล้วเก็บคำเหล่านั้นเข้าตาราง โดยในตารางจะเก็บคู่ศัพท์กับความถี่ของคำศัพท์นั้นเอาไว้ เมื่อเก็บข้อมูลเสร็จแล้วจงพิมพ์จำนวนคำ(ไม่รวมคำซ้ำ) ทั้งหมดออกมา

ข้อนี้เราเขียนโค้ดได้ดังรูป 4.27 โดยให้คำศัพท์แต่ละคำเป็นคีย์ และ ความถี่ของแต่ละคำที่เกิดเป็นวาลูย์

```
1:     import java.util.*;
2:     public class Freq {
3:        private static final Integer ONE = new Integer(1);
4:        public static void main(String args[]) {
5:           Map m = new HashMap();
6:           for (int i=0; i<args.length; i++){
7:               //หาว่าคำนี้มีความถี่เท่าไรในตอนนี้
8:               Integer freq = (Integer)m.get(args[i]);
9:               if(freq == null) //ถ้าไม่มีก็ใส่หนึ่ง
10:                 m.put(args[i], ONE);
11:              else{
12:                 int new = freq.intValue()+1;
13:                 m.put(args[i], new Integer(new));
14:              }
15:          }
16:          System.out.println("มีคำศัพท์" + m.size() + "คำ");
17:          System.out.println(m); // พิมพ์แม็ปออกไปเลย
18:       }
19:    }
```

รูป 4.27 โปรแกรมเก็บค่าความถี่ของคำที่ใส่มาในอินพุต

ตอนที่พิมพ์ออกมาอาจจะพิมพ์ไม่เรียงกันเพราะเราใช้แฮชแม็ป (HashMap) ซึ่งเก็บข้อมูลแบบยัดๆไปไม่เรียงลำดับ (ดูหลักการของแฮชแม็ปได้จากเรื่องตารางแฮช ในตอนท้ายของหนังสือเล่มนี้) ถ้าเราอยากให้ข้อมูลเรียงกันภายในแม็ป ก็ต้องใช้ทรีแม็ป (TreeMap)

ตัวอย่างที่ 4-5

จงสร้างโปรแกรมที่ลูปพิมพ์คีย์ของแม็ป m ออกมาทั้งหมด

ข้อนี้เราลูปด้วยเซตของคีย์ก็พอ โดยมีหลักคือ เราสามารถดึงอิเทอเรเตอร์ออกมาได้เช่นเดียวกับคอลเลคชั่นอื่นๆ เขียนเป็นโค้ดได้ในรูป 4.28

```
1:    ...
2:    Iterator i;
3:    for (i= m.keySet().iterator; i.hasNext(); )
4:          System.out.println(i.next());
5:    ...
```

รูป 4.28 โปรแกรมลูปบนเซตของคีย์

ตัวอย่างที่ 4-6

จงสร้างโปรแกรมที่ลูปพิมพ์คู่ (key, value)ของแม็ป m ออกมาทั้งหมด โดยใช้อิเทอเรเตอร์

เราสามารถใช้อิเทอเรเตอร์ของเอ็นทรี่เซตในการลูปได้ ดังรูปที่ 4.29

```
1:    ...
2:    Iterator i;
3:    for (i= m.entrySet().iterator; i.hasNext(); ){
4:      Map.Entry e = (Map.Entry)i.next();
5:      System.out.println(e.getKey()+ ":"+ e.getValue());
6:    }
7:    ...
```

รูป 4.29 โปรแกรมลูปบนเซตของคู่ (key, value)

ตัวอย่างที่ 4-7

ถ้าเรามีแม็ป m1 กับ m2 จงเขียนโค้ดเพื่อทดสอบว่า m2 เป็น submap ของ m1 หรือไม่

จริงๆข้อนี้ถ้าเรารู้คำสั่งของแม็ปก็ง่ายนิดเดียว โค้ดอยู่ในรูปที่ 4.30

```
1:    ...
2:    if(m1.entrySet().containsAll(m2.entrySet()))
3:    ...
```

รูป 4.30 ทดสอบว่า m2 เป็น submap ของ m1 หรือไม่

ตัวอย่างที่ 4-8

สมมติว่าเรามีข้อมูลของสายลับที่ร่วมประชุมลับ โดยข้อมูลถูกเก็บเป็นคู่ของ (รหัสสายลับ, ชื่อ) นอกจากนี้ยังมีเซตอีกสองเซต เซตหนึ่งใช้เก็บรหัสสายลับของผู้ที่ต้องเข้าร่วมประชุมทุกครั้ง อีกเซตหนึ่งใช้เก็บรหัสสายลับของผู้ที่เราอนุญาตให้เข้าร่วมประชุมได้ แน่นอนว่าผู้ที่ต้องเข้าประชุมก็เป็นพวกที่ได้รับอนุญาตให้เข้าประชุมด้วย จงเขียนโค้ดเพื่อทดสอบว่า การประชุมครั้งนี้มีคนที่ต้องเข้าประชุมขาดไปหรือไม่ และมีคนที่ไม่พึงประสงค์ปะปนเข้ามาด้วยหรือไม่

ข้อนี้เขียนเป็นโค้ดได้ดังรูปที่ 4.31 สมมติให้ Map ที่เก็บคู่ (รหัสสายลับ, ชื่อ) นั้น เป็นตัวแปรที่มีชื่อว่า attendantMap

```
1:    ...
2:    Set attendantCode = attendantMap.keySet();
3:
4:    //ถ้าในที่ประชุม ไม่ได้มีคนที่ต้องการหมดทุกคน
5:    if(!attendantCode.containsAll(requiredSet)){
6:        Set missing = new HashSet(requiredSet);
7:
8:        //เอาคนที่เข้าประชุมออกจากเซตของคนที่ต้องเข้าประชุม ก็จะเหลือคนที่ต้องเข้าแต่ขาด
9:        missing.removeAll(attendantCode);
10:       System.out.println("Missing ="+missing );
11:   }
12:
13:   //ถ้าคนที่อนุญาตให้เข้าประชุมไม่ตรงกับคนที่เข้าประชุมจริง
14:   if(!permissibleSet.containsAll(attendantCode)){
15:       Set illegal = new HashSet(attendantCode);
16:
17:       //เอาคนที่มีสิทธิ์เข้าประชุมออกจากเซตของคนที่เข้าประชุมจริง ถ้าเหลือก็คือมีคนแปลกปลอมเข้ามา
18:       illegal.removeAll(permissibleSet);
19:       System.out.println("Illegal person =" +illegal);
20:   }
21:   ...
```

รูป 4.31 ทดสอบคนที่ขาดประชุมและคนแปลกปลอม

ตัวอย่างที่ 4-9

สมมติว่าเรามีข้อมูลของนักฟุตบอลในประเทศอังกฤษแต่ละคนซึ่งเก็บเป็นคู่ของ (ชื่อนักฟุตบอล, ชื่อผู้จัดการทีมที่เขาอยู่ด้วยนานที่สุด) เราเรียกแม็ปนี้ว่า playerInfo จงเขียนโค้ดเพื่อกระทำการต่อไปนี้

ก. หาเซตของคนที่ไม่ใช่ผู้จัดการทีม (อย่าลืมว่านักฟุตบอลก็เป็นผู้จัดการทีมได้ด้วย)
ข. ลบลูกทีมของผู้จัดการ Alex Ferguson ออกจากฐานข้อมูลนี้ให้หมด
ค. หานักฟุตบอลที่มีผู้จัดการทีมที่เขาอยู่ด้วยนานที่สุดเป็นคนที่ไม่เคยเป็นนักฟุตบอล

ข้อแรกทำได้ดังโค้ดในรูป 4.32

```
1:    ...
2:    Set normal = new HashSet(playerInfo.keySet());
3:
4:    //ลบพวกที่เป็นผู้จัดการออกจากเซตที่เก็บชื่อผู้เล่นทั้งหมด ก็จะเหลือแต่ผู้เล่นที่ไม่เป็นผู้จัดการ
5:    normal.removeAll(playerInfo.values());
6:    ...
```

รูป 4.32 หาเซตของคนที่ไม่ใช่ผู้จัดการทีม

ส่วนการลบลูกทีมของ Alex Ferguson ทิ้งไปนั้น ต้องใช้การลบคนที่ชื่อ “Alex Ferguson” ออกจาก values ของแม็ปนี้ ซึ่งจะทำให้ทั้ง Entry นั้นถูกลบไปเลย ดังโค้ดในรูป 4.33

```
1:    ...
2:    playerInfo.values().removeAll(Collections.singleton
(“Alex Ferguson”));
3:    ...
```

*รูป 4.33 การลบคนที่เคยอยู่กับ **Alex Ferguson** นานๆ ทิ้งไปทั้งหมด*

จะเห็นว่า เราใช้ `Collections.singleton` ซึ่งเป็นเมธอดที่รีเทิร์นเซตซึ่งมีสมาชิกดังตัวที่ระบุออกมา เซตที่ได้นี้ไม่สามารถจะแก้ไขได้

สำหรับข้อสุดท้ายนั้น เราสามารถเขียนได้โดยมีหลักการคือ เอาชื่อนักเตะมาลบออกจากชื่อผู้จัดการ แล้วเราจะเหลือผู้จัดการที่ไม่ได้เป็นนักฟุตบอลเท่านั้น จากนั้นก็เอาแม็ปของสิ่งที่เหลือมาเป็นคำตอบสุดท้าย ดังแสดงในรูป 4.34

```
1:    ...
2:    Map m = new HashMap(playerInfo);
3:    m.values().removeAll(playerInfo.keySet());
4:    Set remain = m.keySet();
5:    ...
```

รูป 4.34 นักฟุตบอลที่ผู้จัดการทีมไม่ได้เป็นนักฟุตบอล

แบบฝึกหัด

1. ถ้าเรามี แม็ป m1 กับ m2 จงเขียนโค้ดบรรทัดเดียวเพื่อหาว่า m1 กับ m2 มี คีย์เหมือนกันหมดหรือไม่
2. ถ้าต้องการรู้คีย์ที่มีทั้งในแม็ป m1 กับ m2 เราควรเขียนโค้ดเพื่อหาคีย์เหล่านั้นอย่างไร
3. ถ้าต้องการเอาคู่ (key, value) ที่แม็ป m2 มีซ้ำกับ m1 ทิ้งออกจาก m2 ไปเลย จะต้องเขียนโค้ดอย่างไร
4. ถ้าในแม็ป m2 เราต้องการเอาสมาชิกที่มีคีย์เหมือนกับใน m1 ทิ้งออกไป จะต้องเขียนโค้ดอย่างไร
5. กำหนดให้มีตัวเลขที่วางเรียงกันดังต่อไปนี้คือ

   1, 2, 3, 4, 5, 6, 7, 8, 9, 0

   a. จงเอาตัวเลขเหล่านี้ใส่ในโครงสร้างข้อมูลที่เราออกแบบเอง ซึ่งถือว่าเป็น Java List ชนิดหนึ่ง โดยให้โครงสร้างข้อมูลของเรามี Constructor method ที่เรียงตัวเลขให้เรียงกันตามชุดตัวเลขที่กำหนดให้ ยังไม่ต้องเขียนเมธอดอื่น

b. จงเขียนเมธอดชื่อ deleteOdd() ของโครงสร้างข้อมูลที่เราสร้างขึ้นในข้อที่แล้ว เพื่อให้ได้ผลลัพธ์เป็น List ที่มีแต่ตัวเลขที่เป็นเลขคู่เท่านั้น

c. จงเขียนเมธอดชื่อ reverseList() ของโครงสร้างข้อมูลที่เราสร้างขึ้น เพื่อให้ได้ผลลัพธ์เป็น List ที่มีตัวเลขที่เรียงกลับด้านเมื่อเทียบกับสิ่งที่อยู่ใน List ต้นฉบับ

d. จงเขียนเมธอดที่ชื่อ hashcode( ) ให้เหมาะสมกับโครงสร้างข้อมูลที่เราสร้างขึ้นมากที่สุด

6. กำหนดให้คลาส Animal เป็นสัตว์ใดๆ ที่มีชื่อสัตว์, ประเภท (คือ Mammal, Bird, Reptile), จำนวนขา

   a. จงเขียนภาษาจาวาเพื่อกำหนดคลาส Animal

   b. จงสร้างคลาส Animals ให้เป็น Collection หนึ่งซึ่งสามารถเก็บ Animal ได้ จงเขียนทุกเมธอดของ Collection นี้ ทำการสมมติเมธอดบางอย่างขึ้นมาได้ในการช่วยเขียน ตามที่เห็นว่าเหมาะสม

   c. จงเขียนเมธอดชื่อ mammalSet( ) ที่ทำการรีเทิร์น Set ของสัตว์ที่เป็น Mammal จากใน Animals

   d. จงเขียนเมธอดในการพิมพ์ element ใน Animals ทุกตัวออกมาให้ครบ

7. กำหนดให้ข้อมูลต่อไปนี้ที่ประกอบด้วยเลขประจำตัวนิสิต และชื่อ

   4423423419 Somchai T.

   4412345678 Sombat B.

   4434321234 Somchai K.

   a. จงเขียนส่วนของโค้ดในการจัดเก็บข้อมูลนิสิตข้างต้น ใส่ในโครงสร้างข้อมูลแบบ Map ที่ใช้ใน Java Collection Framework

   b. จงเขียนส่วนของโค้ดในการค้นหาชื่อนิสิตจากเลขประจำตัว

   c. จงเขียนส่วนของโค้ดในการค้นหาเลขประจำตัวนิสิตจากชื่อนิสิต

8. จงสร้างอาร์เรย์ลิสต์ของ Integer ขึ้นมาแล้วเขียนโค้ดของการเติมสมาชิกลงไปหนึ่งตัว

9. จงสร้าง TreeSet ของ String ขึ้นมา และเขียนโค้ดเติมสมาชิกเข้าไปหนึ่งตัว

10. จงสร้าง HashSet ของ Student ขึ้นมา โดยให้ชื่อของ Student เป็น key และ เกรด เป็น value จงเขียนโค้ดในการเติม Student เข้าไปหนึ่งคน

11. จงเขียนโปรแกรมเพื่อเก็บ String ในอาร์เรย์ลิสต์ โปรแกรมนี้จะเติม String 3 คำตามที่กำหนดไว้จาก input ใส่อาร์เรย์ลิสต์ไป แล้วเอาสมาชิกที่อยู่ที่ตำแหน่งดัชนีที่ 1 ของอาร์เรย์ลิสต์ทิ้งไป หลังจากนั้นก็จะเติมสมาชิกตัวแรกเข้าไปในอาร์เรย์ลิสต์อีกครั้งหนึ่ง สุดท้ายก็พิมพ์สมาชิกทั้งหมดของลิสต์นั้นออกไปหน้าจอ

12. ถ้ามีโค้ดดังนี้

```
1.    ...
2.    LinkedList<String> x = new LinkedList<String>();
3.    x.add(“Shining Finger!!!!”);
4.    Iterator<String> itr = x.iterator();
5.    Integer y = itr.next();
6.    ...
```

ถามว่า โค้ดนี้ผิดตรงไหน อย่างไร

13. จงสร้างลิสต์ ของ Integer ซึ่งสร้างจากอาร์เรย์ลิสต์ขึ้นมา เขียนเมธอดเพื่อเอาเลข 1 ถึง 10 ใส่ในอาร์เรย์ลิสต์นี้ หลังจากนั้นสร้างเซตของอาร์เรย์ลิสต์ที่เก็บ Integer ขึ้นมา แล้วเอาลิสต์ของเรานี้ใส่เซตที่สร้างขึ้น

14. จงทำแบบฝึกหัดในบทของ ลิสต์ สแตก คิว ใหม่ โดยใช้ลิสต์ของจาวาแทน