บทที่ 5

อาร์เรย์ลิสต์กับลิงค์ลิสต์ในจาวา

จาวานั้นมีโครงสร้างข้อมูลแบบลิสต์ให้เรียกใช้อยู่ในไลบรารี่ของมันเองอยู่แล้ว โดยมีสองรูปแบบคือ อาร์เรย์ลิสต์ และ ลิงค์ลิสต์ ซึ่งทั้งคู่อิมพลีเม้นท์ลิสต์อินเตอร์เฟส ซึ่งก็หมายความว่าทั้งคู่นั้นใช้เมธอดจากลิสต์อินเตอร์เฟสได้ทั้งหมด แต่โครงสร้างการเก็บข้อมูลภายในนั้นต่างกัน นอกจากนี้ยังมีเมธอดที่เพิ่มจากลิสต์อินเตอร์เฟสอีกด้วย (แต่การจะใช้งานเมธอดที่ไม่มีในลิสต์อินเตอร์เฟสนั้นทำให้เราต้องปักหลักใช้คลาสที่มีเมธอดนั้นอย่างเดียว) ในที่นี้จะขอกล่าวถึงอาร์เรย์ลิสต์เสียก่อน

อาร์เรย์ลิสต์(ArrayList)

อาร์เรย์ลิสต์นั้นมีลักษณะของทั้งอาร์เรย์และลิสต์อยู่ในตัวเดียวกัน เราสามารถใช้ค่าดัชนีเพื่อเข้าถึงสมาชิกในอาร์เรย์ลิสต์แต่ละตัวได้ เช่นเดียวกับการใช้ดัชนีของอาร์เรย์ อาร์เรย์ลิสต์นั้นมีเมธอดส่วนใหญ่ทำงานเหมือนกับที่นิยามไว้ในลิสต์อินเตอร์เฟส และก็มีเมธอดซึ่งไม่มีในลิสต์อินเตอร์เฟส นั่นคือ เมธอด clone เมธอด ensureCapacity เมธอด trimToSize และเมธอด removeRange แต่จุดที่เด่นที่สุดของอาร์เรย์ลิสต์เห็นจะเป็นคุณสมบัติที่ว่า

- ใส่ได้แต่ object เท่านั้น และ
- ขยายขนาดเองได้โดยอัตโนมัติ

โครงสร้างคลาสของอาร์เรย์ลิสต์

อาร์เรย์ลิสต์นั้นมีโครงสร้างคลาสดังแสดงในรูป 5.1 ซึ่งเราจะเห็นได้ว่าอิมพลีเม้นท์อยู่หลายอินเตอร์เฟสด้วยกัน คลาสอาร์เรย์ลิสต์นั้นเป็นลูกของแอบแสตร็กลิสต์อีกทีหนึ่ง มีฟิลด์ข้างใน

สองตัว ตัวหนึ่งเป็นอาร์เรย์ที่เราจะใช้เก็บของนั่นเอง ส่วนอีกตัวหนึ่งเก็บจำนวนสมาชิก ดังนั้นเราสามารถสรุปได้ง่ายๆว่าอาร์เรย์ลิสต์คืออาร์เรย์ของออบเจ็กต์นั่นเอง

```
1:    public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Clonable, Serializable{
2:
3:    //transient หมายถึง ไม่เซฟตัวอาร์เรย์ตอนทำ serialization แต่เซฟสมาชิกนะ
4:    private transient E[] elementData;
5:
6:    //จำนวนสมาชิก
7:    private int size;
```

รูป 5.1 โครงสร้างคลาสและฟิลด์ของอาร์เรย์ลิสต์

คอนสตรัคเตอร์ของอาร์เรย์ลิสต์

คลาสนี้จะมีคอนสตรัคเตอร์อยู่สามชนิด ดังแสดงในรูป 5.2 ถ้าเราไม่บอกขนาดของลิสต์ไป คอนสตรัคเตอร์จะสร้างอาร์เรย์ลิสต์ที่จุสมาชิกได้สิบตัวเท่านั้น (แต่อย่าลืมว่าขนาดนั้นขยายได้อยู่แล้ว) รูป 5.3 แสดงตัวอย่างการเรียกใช้คอนสตรัคเตอร์อย่างง่าย

แต่จากรูป 5.2 ยังมีคอนสตรัคเตอร์อีกแบบหนึ่ง ซึ่งสร้างอาร์เรย์ลิสต์ขึ้นจากการก็อปปี้สมาชิกจากคอลเลคชั่นตัวอื่น ถ้าเรามี womanList ซึ่งจุของประเภท Woman ซึ่งเป็นสับคลาสของ People เราสามารถสร้าง อาร์เรย์ลิสต์ของ People ได้จาก womanList ดังรูป 5.4

จากรูป 5.4 แม้ว่าจะเป็นการสร้างอาร์เรย์ลิสต์ขึ้นมาใหม่ แต่ภายในนั้น สมาชิกของอาร์เรย์ลิสต์แต่ละตัวก็ยังเป็นเพียงพอยต์เตอร์ไปยังของใน womanList เท่านั้น นี่ถือว่าเป็น shallow copy แบบหนึ่ง วิธีการสร้างอาร์เรย์ลิสต์ขึ้นมาจากลิสต์ที่มีอยู่แล้ว อีกแบบหนึ่ง คือการใช้เมธอด clone() ดังแสดงในรูป 5.5

หลายคนอาจจะยังคงว่า shallow copy นั้นคืออะไรกันแน่ ผมจะขออธิบายไว้เลยว่า shallow copy นั้นคือการก็อปปี้ออบเจ็กต์และพอยต์เตอร์ที่ชี้ไปยังออบเจ็กต์หนึ่งๆขึ้นมาต่างหากจากตัวต้นฉบับ แต่ถ้าภายในของออบเจ็กต์ที่ถูกก็อปปี้มานั้นมีออบเจ็กต์อื่นอยู่ พอยต์เตอร์ของออบเจ็กต์ตัวนี้จะไม่ถูกก็อปปี้แยกไป แต่จะยังชี้ไปยังที่เดิม

```
1:    //สร้างอาร์เรย์ของวัตถุขึ้นมา ให้มีความจุเท่ากับที่ระบุไว้
2:    //IllegalArgumantException ถ้า initialCapacity <0
3:    public ArrayList (int initialCapacity)){
4:       elementData =new Object [initialCapacity];
5:    }
6:
7:    //สร้างอาร์เรย์ของวัตถุให้มีความจุ 10
8:    public ArrayList (){
9:       this(10);
10:   }
11:
12:   //สร้างอาร์เรย์ใหม่ให้ใหญ่กว่าของเดิม 10% แล้วก็อปปี้ของจาก C ลงไปหมดทุกตัว
13:   // ลำดับของสมาชิกก็เอาตามลำดับสมาชิกของ c
14:   // Worst time = O(n)
15:   // ชนิดของสมาชิกจาก c ต้องเป็นชนิดเดียวกับในอาร์เรย์ลิสต์ หรือไม่ก็ต้องเป็นสับคลาส
16:   public ArrayList(Collection<? extends E> c){
17:      this((c.size()*110)/100); // โตขึ้น 10%
18:      Iterator i = c.iterator();
19:      while(i.hasNext())
20:         elementData[size++] = i.next();
21:   }
```

รูป 5.2 คอนสตรัคเตอร์ของอาร์เรย์ลิสต์

```
1:    //บรรทัดล่างนี้คือตัวอย่างการใช้งาน
2:    ArrayList<String> name = new ArrayList<String>(100);
```

รูป 5.3 ตัวอย่างการใช้งานคอนสตรัคเตอร์ของอาร์เรย์ลิสต์

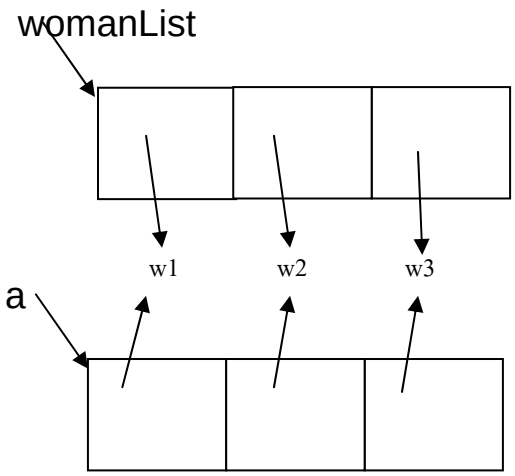
```
1:    ArrayList<People>  a = new
ArrayList<People>(womanList);
```

รูป 5.4 การสร้างลิสต์ใหม่โดยก็อปปี้จากของเดิม

```
1:    ArrayList<People> a =
(ArrayList<People>)womanList.clone();
```

รูป 5.5 การสร้างลิสต์ใหม่โดยใช้เมธอดโคลน

การใช้คอนสตรัคเตอร์และการโคลนจะไม่เหมือนกับการใช้ a = womanList; เพราะการใช้ a = womanList จะให้ a ชี้ไปที่ womanList เลย รูป 5.6 แสดงสภาวะของออบเจ็กต์ที่เกิดจากโค้ดโปรแกรมของรูป 5.5 โดย w แต่ละตัวคือตัวเนื้อที่ที่เก็บข้อมูลของสมาชิก



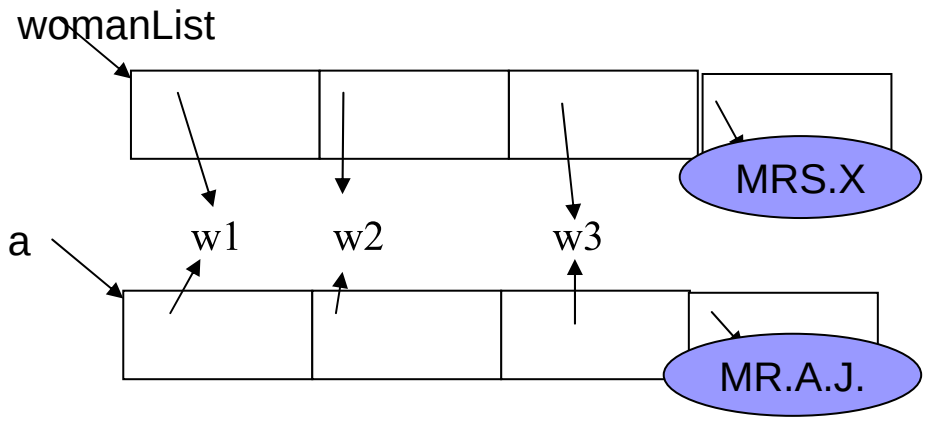
รูป 5.6 สภาพของลิสต์เก่าและใหม่จากการโคลน

จากรูป 5.6 ถ้าเราเรียกใช้เมธอดดังนี้คือ

- womanList.add(new People("MRS.X")); กับ
- a.add(new People("MR.A.J."))

จะได้ผลลัพธ์เป็นดังรูป 5.7 ซึ่งจะเห็นได้ว่า womanList กับ a นั้นเป็นคนละลิสต์กัน เพียงแต่ว่าสมาชิกภายในอาจมีลิงค์ไปที่วัตถุชิ้นเดียวกันได้

แต่การสร้างลิสต์ใหม่จากลิสต์ที่มีอยู่แล้วด้วยคอนสตรัคเตอร์ กับ ด้วยการโคลนนั้น มีข้อแตกต่างกันอย่างหนึ่งคือ การโคลนจะได้จำนวนช่องของอาร์เรย์เท่าเดิม ในขณะที่การใช้คอนสตรัคเตอร์นั้นเราจะได้อาร์เรย์ที่มีขนาดใหญ่กว่าเดิมสิบเปอร์เซ็นต์ ดังนั้นเวลาใช้งานจึงควรรู้ถึงข้อนี้ด้วย



รูป 5.7 การเติมสมาชิกใหม่เข้าไปในลิสต์ต้นฉบับและลิสต์ที่เกิดจากการ โคลน

เมธอดต่างๆของอาร์เรย์ลิสต์

เมธอดของอาร์เรย์ลิสต์นั้นส่วนใหญ่เป็นเมธอดที่นิยามไว้ในลิสต์อินเตอร์เฟสอยู่แล้ว โดยมีเมธอดที่สำคัญดังต่อไปนี้

1. public boolean add(E element)

เป็นเมธอดที่ทำการเอา element ใส่ท้ายอาร์เรย์ลิสต์ โดยมีเวลากรณีที่แย่ที่สุดเป็น O(n) และมีเวลาโดยเฉลี่ยเป็นค่าคงที่ เมธอดนี้รีเทิร์น true เมื่อ element ได้ถูกใส่ลงไปในอาร์เรย์ลิสต์จริงๆ (แบบนี้ก็ true ตลอดเพราะว่าแต่ละครั้งก็สามารถใส่ของลงไปในอาร์เรย์ลิสต์ได้ เพราะยังไงอาร์เรย์ลิสต์ก็ขยายขนาดได้เองอยู่แล้ว)

2. public Object clone()

รีเทิร์น shallow copy ของตัวอาร์เรย์ดังที่ได้อธิบายไปแล้วข้างต้น (ก็อปปี้ตัวอาร์เรย์แต่ไม่ก็อปปี้สมาชิกภายใน) เมธอดนี้ไม่มีในลิสต์อินเตอร์เฟส

3. public void ensureCapacity(int minCapacity)

ขยายอาร์เรย์ถ้าจำเป็น เพื่อให้สามารถจุจำนวนสมาชิกขนาดที่ระบุโดย minCapacity ได้ เมธอดนี้ไม่มีในลิสต์อินเตอร์เฟส

4. public void trimToSize()

ลดขนาดของอาร์เรย์ลิสต์ให้เหลือจำนวนช่องอาร์เรย์เท่ากับจำนวนสมาชิกที่มีอยู่ในปัจจุบัน เมธอดนี้ไม่มีในลิสต์อินเตอร์เฟส

5. public E get(int index)

เมธอดนี้รีเทิร์นสมาชิก ณ ตำแหน่งที่บอกด้วย index ออกมาเป็นคำตอบ โดยจะมีการ throw indexOutOfBoundsException ถ้า size() < = index, หรือ index < 0 ซึ่งเมธอดนี้ก็เปรียบได้กับการเข้าดูสมาชิกของอาร์เรย์โดยอาศัยตัวเลข index นั่นเอง

6. public E set(int index, E element)

เมธอดนี้ทำการแทนที่สมาชิกตัวที่ index ด้วย element และรีเทิร์นสมาชิกเก่าที่เก็บที่ index นี้ออกมาเป็นคำตอบ โดยจะ throw indexOutOfBoundsException ถ้า size() < =index หรือ index < 0 เมธอดนี้มีการใช้เวลามากที่สุดเป็นค่าคงที่เพราะเราสามารถเข้าถึงสมาชิกของอาร์เรย์ลิสต์ตัวที่ระบุด้วย index ได้ทันที

7. public void add(int index, E element)

เมธอดนี้ทำการแทรก element เข้าไป ณ ตำแหน่ง index ดังนั้นสมาชิกตัวอื่นที่อยู่หลังตำแหน่งที่ index ต้องเลื่อนตำแหน่งออกไป เมธอดนี้มีเวลาในการทำงานมากที่สุดเป็น O(n) เพราะจะเสียเวลาเลื่อนสมาชิกนั่นเอง การเรียกใช้เมธอดนี้อาจเกิดการ throw indexOutOfBoundsException ได้ ถ้า size() < index หรือ index < 0

8. public E remove(int index)

เมธอดนี้จะทำการเอาสมาชิก ณ ตำแหน่ง index ออกไป ดังนั้นสมาชิกทางขวาของตำแหน่ง index นั้นต้องเลื่อนมาแทนที่ เมธอดนี้รีเทิร์นสมาชิกตัวที่เอาออกไปนั่นเอง และ throw indexOutOfBoundsException ถ้า size() < = index หรือ index < 0

9. protected void removeRange(int fromIndex, int toIndex)

เมธอดนี้เอาของในอาร์เรย์ลิสต์ที่อยู่ในดัชนีตั้งแต่ fromIndex ถึง toIndex (นับรวมสมาชิกที่ตำแหน่ง fromIndex แต่ไม่รวมสมาชิกที่ตำแหน่ง toIndex) ออกจากอาร์เรย์ลิสต์ไป สมาชิก

toIndex และสมาชิกที่อยู่ต่อหลังมันจะเลื่อนตำแหน่งมาเติมช่องว่างในอาร์เรย์เอง ถ้า toIndex==fromIndex ก็จะไม่มีอะไรเกิดขึ้น เมธอดนี้เป็นอีกเมธอดหนึ่งที่ไม่มีในลิสต์อินเตอร์เฟส

10. public int indexOf(Object element)

เมธอดนี้ทำการหาค่าที่เหมือนกับ element เป็นตัวแรกในอาร์เรย์ลิสต์ โดยจะรีเทิร์น ดัชนีของค่านั้นในอาร์เรย์ลิสต์ หรือ -1 ถ้าหาไม่เจอ เมธอดนี้ทำการเปรียบเทียบค่าของ element ว่าเหมือนกับสมาชิกอื่นของอาร์เรย์ลิสต์หรือไม่ด้วยเมธอด equals() ส่วนด้านเวลาในการทำงานนั้น เมธอดนี้ใช้เวลาทำงานกรณีที่นานที่สุดคือ O(n) ซึ่งเกิดจากการไล่หาของจากทั้งลิสต์แล้วหาไม่เจอนั่นเอง

โค้ดของเมธอดที่น่าสนใจ

โค้ดของเมธอด add ซึ่งเอาวัตถุใส่ท้ายอาร์เรย์ลิสต์นั้นอยู่ในรูป 5.8 โดยเราจะเห็นว่า จะต้องมีการขยายอาร์เรย์ก่อนถ้าจำเป็นด้วยการเรียกใช้เมธอด ensureCapacity นอกจากนี้เมธอดนี้ยังรีเทิร์น true เสมออีกด้วย ส่วนโค้ดของ ensureCapacity นั้นอยู่ในรูป 5.9

```
1:    public boolean add(E element){
2:     ensureCapacity(size+1); //ขยายอาร์เรย์ถ้าจำเป็น
3:     elementData[size++] = element;
4:     return true;
5:    }
```

รูป 5.8 โค้ดของการเติมของเข้าไปท้ายอาร์เรย์ลิสต์

จากรูป 5.9 จะเห็นว่ามีตัวแปรชื่อ modCount อยู่ด้วย โดยตัวแปรนี้ มีในคลาส ArrayList LinkedList TreeMap TreeSet HashMap และ HashSet สำหรับตัวแปร modCount ในอาร์เรย์ลิสต์นั้น ได้รับสืบทอดมาจากคลาสแอบแสตร็กลิสต์ ซึ่งค่าของตัวแปรนี้จะเปลี่ยนเมื่อมีการ insert หรือ remove จาก ArrayList นี้

อิเทอเรเตอร์แต่ละตัวก็มีตัวแปร expectedModCount อยู่ด้วยต่างหาก ซึ่งค่าเริ่มต้นของ expectedModCount ก็คือค่าของ modCount นั่นเอง ค่าของ modCount และ expectedModCount

จะเปลี่ยนเมื่อ อิเทอเรเตอร์ตัวนั้นเปลี่ยนของในคอลเลคชั่น เช่นเรียกใช้เมธอด itr.remove() ถ้าสองฟิลด์นี้ไม่เท่ากันเมื่อไร แสดงว่ามีการเปลี่ยนคอลเลคชั่นด้วยเมธอดที่ไม่ใช่ของอิเทอเรเตอร์ระหว่างที่อิเทอเรเตอร์มีการใช้งานอยู่ เช่นเปลี่ยนโดยเทรดอื่น ซึ่งอาจทำให้เสียสถานะที่ดีได้ เช่น ถ้าเราทำการลบของออกจากอาร์เรย์ลิสต์โดยไม่ใช้อิเทอเรเตอร์ อาจทำให้ของในตำแหน่งที่อิเทอเรเตอร์จะดูต่อไปออกจากอาร์เรย์ลิสต์ไปแล้ว ยามเมื่อเราเรียก next() จากอิเทอเรเตอร์ก็จะกลายเป็นว่าอิเทอเรเตอร์จะเลื่อนไปตรงที่เราไม่ได้คาดหวังไว้ ซึ่งอาจจะเป็นการดูเลยอาร์เรย์ลิสต์ไปเลยก็ได้ รูป 5.10 แสดงโค้ดที่ถูกเรียกใช้เพื่อตรวจสอบ modCount ทุกครั้งที่มีการเปลี่ยนแปลงของลิสต์ด้วยเมธอดของอิเทอเรเตอร์

```
1:    public void ensureCapacity(int min)
2:    {
3:       modcount++; //เดี๋ยวอธิบายตัวนี้
4:       int oldCapacity =elementData.length;
5:       if(min > oldCapacity){ //ความจุใหม่ใหญ่กว่าเก่า
6:          E oldData[] = elementData;
7:          int newCapacity =(oldCapacity*3)/2+1; //ให้ขยาย 50%
8:          if(newCapacity<minCapacity) //ถ้ายังขยายไม่พอ
9:             newCapacity =min; // ก็ให้ขนาดตามอินพุตไปเลย
10:         elementData =(E)new Object[newCapacity];
11:       System.arrayCopy(oldData,0,elementData,0,size);
12:      }
13:   }
```

รูป 5.9 โค้ดของ ensureCapacity

```
1:    if (modCount!=expectedModCount)
2:        throw new ConcurrentModificationException();
```

รูป 5.10 การตรวจสอบค่าของ modCount และ expectedModCount

ตัวอย่างที่ 5-1

ตัวอย่างนี้แสดงการใช้อิเทอเรเตอร์อย่างไม่ถูกต้อง โดยมีโค้ดดังรูป 5.11 ซึ่งจะเห็นได้ว่า ในตอนที่มีการใช้เมธอด add นั้น modCount จะมีค่าเพิ่มขึ้น แต่ expectedModCount จะยังมีค่าเท่าเดิม ทำให้การตรวจสอบค่าของตัวแปรทั้งสองในยามที่เรียกใช้เมธอด next นั้นไม่ผ่าน

```
1:    public ModCountDriver() {
2:     ArrayList list =new ArrayList();
3:     list.add ("yes");
4:     Iterator itr = list.iterator(); //สร้างอิเทอรเรเตอร์
5:     list.add ("good"); //ใช้เมธอดอื่นเปลี่ยนของในอาร์เรย์หลังสร้างอิเทอร์เรเตอร์
6:     itr.next(); //โปรแกรมจึงมา throw exception ตรงนี้
7:    }
```

รูป 5.11 โปรแกรมหาที่ใช้อิเทอร์เรเตอร์ไม่ถูกต้อง

คราวนี้เรามาวิเคราะห์เวลาในการทำงานของ add กันบ้าง เวลาที่จะใช้มากที่สุดก็คือเวลาของการก็อปปี้อาร์เรย์เมื่อ ensureCapacity ถูกเรียกใช้ ทำให้มี big O เป็น O(n)

ส่วนเวลาโดยเฉลี่ยนั้น เราคิดได้จากข้อมูลต่อไปนี้

- หลังจากการขยายอาร์เรย์ไปครั้งหนึ่งแล้ว ต้องเรียก add ไป n/3 ครั้งก่อน ถึงจะมีการก็อปปี้อาร์เรย์เกิดขึ้นอีกทีหนึ่ง ทั้งนี้เพราะการขยายอาร์เรย์นั้นขยายขึ้นเพียง 50 เปอร์เซ็นต์
- ฉะนั้น ต้องเรียก add ไป n/3+1 ครั้ง จึงจะเกิดการก็อปปี้ n สมาชิก
- จำนวนการก็อปปี้ ต่อการเรียก add หนึ่งครั้งจึงเฉลี่ยเป็น $\frac{n}{\left(\frac{n}{3}\right)+1}$ นั่นคือ ประมาณ 3 ครั้ง

ดังนั้นเวลาโดยเฉลี่ยของการ add จึงถือเป็นค่าคงที่

เมธอดที่น่าสนใจอีกเมธอดหนึ่งของอาร์เรย์ลิสต์คือเมธอด clone ซึ่งอยู่ในรูป 5.12 ซึ่งโค้ดก็ได้แสดงให้เห็นว่า shallow copy นั้นเป็นอย่างไร

รายละเอียดของเมธอดอื่นนั้นไม่จำเป็นสำหรับผู้นำคลาสนี้ไปใช้งาน เพราะเมธอดส่วนใหญ่ทำงานตามที่นิยามไว้ในลิสต์อินเตอร์เฟสอยู่แล้ว ฉะนั้นจึงไม่กล่าวถึงในที่นี้ จากนี้ไปจะเป็นตัวอย่างต่างๆสำหรับการนำคลาสนี้ไปใช้งาน

```
1:     public Object clone(){
2:      try{
3:        ArrayList v =(ArrayList)super.clone();
4:        v.elementData =new Object[size];
5:        System.arraycopy(elementData, 0, v.elementData, 0, size);
6:        v.modCount =0;
7:      }catch(CloneNotSupportedException e){
8:        throw new InternalError();
9:      }
10:    }
```

รูป 5.12 เมธอดโคลน

ตัวอย่างที่ 5-2
จงเขียนเมธอดซึ่งทำสิ่งต่างๆตามลำดับดังต่อไปนี้

- สร้าง ArrayList ซึ่งมี n สมาชิก
- ลูป n ครั้ง ใส่สมาชิก i ซึ่งเป็น Double โดย i เริ่มจาก 0 ลงไปท้ายลิสต์
- ใส่ new Double (7.8) ที่ index ที่ n/2
- เอาสมาชิกที่ตำแหน่ง 2n / 3 ทิ้งไป
- เอา 2.5 คูณกับสมาชิกตัวกลางของลิสต์
- พริ้นต์ค่าภายในลิสต์ทั้งหมดออกมา

รูป 5.13 แสดงโค้ดของเมธอดตามที่ระบุนี้ แต่เราก็ยังอาจเขียนได้ง่ายกว่าในรูป 5.13 นิดหน่อย ถ้าเราใช้ไทป์แบบมีพารามิเตอร์ ถ้าในบรรทัดที่สามเราสร้างลิสต์โดยใช้

ArrayList<Double> myList = **new** ArrayList (n);

ต่อมาเมื่อเราเขียนโค้ดของบรรทัดที่สิบ เราจะไม่จำเป็นต้องทำการเปลี่ยนไทป์ของข้อมูลให้เป็น Double อีก ดังนั้นเราสามารถใช้

double d = (myList.get (n / 2).doubleValue( )) * 2.5;

ได้โดยไม่ต้องเปลี่ยนไทป์อีก

```
1:    public void processInput (String s) {
2:      int n = Integer.parseInt (s);
3:      ArrayList myList = new ArrayList (n); //สร้างอาร์เรย์ลิสต์
4:      for (int i = 0; i<n; i++)
5:        myList.add (new Double (i)); //ลูปใส่ของเข้าลิสต์
6:      myList.add (n / 2, new Double (7.8)); //ใส่ 7.8 ที่ตำแหน่ง n/2
7:      myList.remove (2 * n / 3); //เอาสมาชิกที่ตำแหน่ง 2n/3 ทิ้งไป
8:
9:      //ต่อไปเอา 2.5คูณกับสมาชิกตัวกลาง
10:     double d = ((Double)myList.get (n / 2)).doubleValue() * 2.5;
11:     myList.set (n / 2, new Double (d));
12:
13:     //แล้วพิมพ์ผลลัพธ์ออกมา
14:     gui.println (myList) ;
15:   }
```

รูป 5.13 โปรแกรมที่ทำตามข้อระบุของตัวอย่างที่ 5-2

ตัวอย่างที่ 5-3

สมมติว่าเรามีอาร์เรย์ลิสต์ซึ่งเก็บ Integer Object เอาไว้ จงเขียนโปรแกรมเพื่อหาผลรวมของตัวเลขทุกตัวที่เก็บด้วยอาร์เรย์ลิสต์นี้

ตัวอย่างนี้ไม่ยากนัก เราสามารถใช้การลูปอย่างง่ายๆในการแก้ปัญหาข้อนี้ได้ และเพราะเราใช้อาร์เรย์ลิสต์ เราจึงเข้าถึงสมาชิกได้ทีละตัวโดยไม่ต้องใช้อิเทอเรเตอร์ด้วยซ้ำ โค้ดของอาร์เรย์ลิสต์จึงเป็นดังรูป 5.14

แต่เวลาที่เราใช้อาร์เรย์ลิสต์จริงๆนั้น อาจใช้กับข้อมูลที่ถือว่าเป็นลิสต์ทั่วไป ดังนั้นถ้าเราเขียนโค้ดโดยใช้ประโยชน์จากค่าดัชนีของอาร์เรย์ลิสต์ จะเป็นการจำกัดไม่ให้เราสามารถไปใช้อิมพลีเม้นเทชั่นของลิสต์แบบอื่น เช่น ลิงค์ลิสต์ ได้อย่างมีประสิทธิภาพ (get(i) ของลิงค์ลิสต์นั้นกินเวลามากเพราะต้องนับจากหัวลิสต์ใหม่ทุกครั้ง) ดังนั้น ถ้าจะให้โค้ดนี้ใช้งานได้กับลิสต์ทั่วไปได้มีประสิทธิภาพ เราจะต้องใช้อิเทอเรเตอร์ ในที่นี้ได้แสดงไว้ในรูป 5.15

```
1:    int sum = 0;
2:    for (int i = 0;  i < list.size();  i++)
3:       sum += list.get(i).intValue( );
```

รูป 5.14 การหาผลรวมของทุกค่าในอาร์เรย์ลิสต์โดยใช้ประโยชน์จากการหา ดัชนี ที่รวดเร็ว

```
1:    int sum = 0;
2:    ListIterator<Integer> i = list.listIterator();
3:    while (i.hasNext())
4:       sum += i.next().intValue();
```

รูป 5.15 การหาผลรวมของทุกค่าในลิสต์โดยใช้อิเทอเรเตอร์

ตัวอย่างที่ 5-4

สมมติว่าเรามีโค้ดอยู่ดังรูป 5.16

```
1:    ArrayList names = new ArrayList();
2:    names.add( "Rana" );
3:    names.add( "Ranafey" );
4:    names.add( "Prettier" );
5:    names.add( "Mark" );
6:    for ( int i = 0; i < names.size(); i++ )
7:    {
8:      System.out.println( names.get( i ) );
9:    }
```

รูป 5.16 โค้ดที่ผิด หรือ ไม่ผิดกันแน่

จงหาว่าโค้ดนี้มีส่วนที่ผิดหรือไม่ อย่างไร

ในตัวอย่างนี้ ดูเผินๆจะเหมือนกับว่าโค้ดผิดในบรรทัดที่เจ็ด เพราะไม่มีการเปลี่ยนไทป์ของข้อมูลกลับเป็นสตริง แต่จริงๆแล้วโค้ดนี้ไม่ผิด เพราะว่าแม้ว่าจะไม่ได้เปลี่ยนไทป์ เมธอด println ก็ยอมรับข้อมูลที่เป็นออบเจ็กต์อยู่แล้ว

จากตัวอย่างนี้ทำให้ได้ข้อสังเกตที่ดีว่า ถ้าเมธอดรองรับไทป์ที่เป็นออบเจ็กต์ เราก็ไม่ต้องกังวลเรื่องการเปลี่ยนไทป์ให้ยุ่งยาก แต่ถ้ากำหนดไทป์ได้ตามจาวา 1.5 ก็จะทำให้อุ่นใจกว่ามาก

ตัวอย่างที่ 5-5

จงสร้างโปรแกรมที่สร้างสับลิสต์จากลิสต์ในรูป 5.16 ขึ้นมา โดยให้สับลิสต์นั้นประกอบไปด้วยสมาชิกจากดัชนีที่ 1 และ 2 เท่านั้น จากนั้นเรียงค่าในสับลิสต์นี้ตามลำดับในพจนานุกรม จากนั้นก็เรียงกลับด้าน แล้วท้ายสุดก็ให้สลับสมาชิกในสับลิสต์นี้แบบสุ่ม

สำหรับตัวอย่างนี้เราสามารถเขียนโค้ดได้ดังรูป 5.17 โดยสามารถใช้เมธอดต่างๆจากคอลเลคชั่นคลาสได้เลย (คอลเลคชั่นคลาสนี้เป็นคนละตัวกับคอลเลคชั่นอินเตอร์เฟส จะสังเกตได้ว่าในชื่อภาษาอังกฤษนั้น คอลเลคชั่นคลาสมีตัว s อยู่ด้วย อย่างไรก็ดี คอลเลคชั่นคลาสนั้นมีไว้เพื่อการจัดการกับคอลเลคชั่นนั่นเอง ดังนั้นจึงสามารถใช้จัดการกับลิสต์ได้ด้วย) การ sort กับ การ reverse นั้นเห็นได้ชัดว่าทำอย่างไร ส่วนการ shuffle นั้นเมธอดรับค่าแรกเป็นตัวลิสต์และค่าที่สองเป็นตัวกำหนดรูปแบบของการสุ่มค่านั่นเอง

```
1:        List listSub = listA.subList(1,3);
2:        Collections.sort(listSub);
3:        Collections.reverse(listSub);
4:        Collections.shuffle(listA, new Random());
```

รูป 5.17 การใช้คอลเลคชั่นคลาส

ตัวอย่างที่ 5-6

จงสร้างคลาสที่มีอาร์เรย์ลิสต์ที่สมาชิกในลิสต์แต่ละตัวใช้เก็บ หลักๆหนึ่งของจำนวนเต็มที่ยาวมากๆ และจงสร้าง

- คอนสตรัคเตอร์ของคลาสนี้ ที่รับสตริงเข้ามา แล้วเปลี่ยนแต่ละหลักเป็นตัวเลขเพื่อเอาใส่ในอาร์เรย์ลิสต์ของเรา แต่หลักที่ไม่ใช่ตัวเลขก็ตัดทิ้งไปเลย ให้ throw NullPointerException ถ้าสตริงเป็น null
- เมธอด toString ซึ่งรีเทิร์นสตริงที่แทนค่าที่เก็บไว้ในอาร์เรย์ลิสต์
- เมธอด add ซึ่งบวกตัวเลขที่เก็บในอาร์เรย์ลิสต์สองตัวเข้าด้วยกัน ให้ throw NullPointerException ถ้าตัวที่มาบวกด้วยนั้นเป็น null

ข้อนี้เราสามารถทำได้โดยกำหนดคลาสขึ้นมา ให้ชื่อว่าคลาส VeryLongInt ซึ่งผมจะให้ฟิลด์ในคลาสนี้เป็นอาร์เรย์ลิสต์ที่เก็บจำนวนเต็มนั่นเอง คลาสนี้และเมธอดทั้งหมดอยู่ในรูป 5.18

```
5:    public class VeryLongInt{
6:      protected ArrayList<Integer>digits;
7:      public VeryLongInt(String s){
8:        final char LOWEST_DIGIT_CHAR = '0';
9:        digits = new ArrayList<Integer>(s.length());
10:       char c;
11:       int digit;
12:       for(int i = 0; i<s.length(); i++){
13:         c = s.charAt(i);
14:         if(Character.isDigit(c)){
15:           digit = c - LOWEST_DIGIT_CHAR;
16:           digits.add(digit); //จาวา 1.5 เปลี่ยนไทป์เอง        }
17:         }
18:       }
19:     }
20:
21:     public String toString(){
22:       return digits.toString();//เรียกใช้เมธอดของอาร์เรย์ลิสต์
23:     }
24:
25:     public void add(VeryLongInt otherVeryLong){
26:      final int BASE = 10;
27:       int largerSize, partialSum, carry = 0;
28:       ArrayList<Integer> sumDigits= new
29:                             ArrayList<Integer>();
30:       if(digits.size()>otherVeryLong.digits.size())
31:         largerSize= digits.size();
32:       else
33:         largerSize = otherVeryLong.digits.size();
34:       for(int i=0; i<largerSize; i++){
35:         partialSum =
36:         least(i) + otherVeryLong.least(i) + carry;
37:         carry = partialSum / BASE ;
38:         sumDigits.add(partialSum % BASE);
39:       }
40:       if(carry==1)
41:         sumDigits.add(carry); //ไงๆหลักสูงสุดก็มี carry ได้ค่าเดียว
42:       Collections.reverse(sumDigits);
43:       digits = sumDigits;
44:     }
45:
46:     protected int least(int i){
47:       if(i>=digits.size())
48:         return 0;
49:       return digits.get(digits.size()-i-1);
50:     }
```

รูป 5.18 โค้ดของการสร้างการบวกเลขตามตัวอย่าง 5-6

สำหรับคอนสตรัคเตอร์นั้น จะมีค่าเวลาแปรตามความยาวของสตริงที่เป็นอินพุตนั่นเอง โดยจะลูปเช็คตัวอักษรทีละตัว ถ้าตัวอักษรนั้นเป็นตัวเลข ให้ลบด้วย '0' เพื่อให้ได้ค่าแท้จริงออกมา หลังจากนั้นจึงเอาเลขที่ได้ใส่ลงในอาร์เรย์ลิสต์ ส่วนเมธอด add ที่คอนสตรัคเตอร์ใช้ภายในนั้นจะมีเวลาโดยเฉลี่ยและเวลาในกรณีแย่สุดเป็นค่าคงที่เพราะอาร์เรย์นี้ไม่ต้องมีการขยายขนาดในตอนใช้คอนสตรัคเตอร์นี้แล้ว ดังนั้นจึงส่งผลให้เวลาของการลูปทั้งกรณีเฉลี่ยและกรณีแย่สุดเป็น O(n) เมื่อ n เป็นขนาดของสตริงนั่นเอง

เมธอด toString นั้นง่ายเพราะสามารถเรียกใช้เมธอดของอาร์เรย์ลิสต์ได้เลย ซึ่ง toString ของอาร์เรย์นั้นกินเวลา O(n) เมื่อ n เป็นขนาดของอาร์เรย์นั่นเอง

สำหรับเมธอด add นั้นมีเวลาในกรณีแย่สุดคือ O(จำนวนหลักเลขของตัวที่มีจำนวนหลักมากที่สุด) เราทำการบวกทีละหลัก เริ่มจากหลักหน่วย (เหมือนบวกเลขปกติ) เอาผลที่บวกแต่ละหลักนั้นใส่อาร์เรย์ลิสต์อีกตัวที่ชื่อว่า sumDigits ตัวอย่างเช่น ถ้าเรามี 135 กับ 79 เราจะได้ผลใน sumDigits เป็น 412 เพราะว่าหลักหน่วยถูกนำไปใส่ sumDigits เสียก่อน ดังนั้นต้องมีการรีเวิร์สสมาชิกใน sumDigits ซึ่งจะได้ 214 เป็นคำตอบที่แท้จริง

ภายในเมธอด add นี้ มีเมธอด least ซึ่งใช้ค้นหาหลักที่ i (เริ่มจากหลักหน่วย) อยู่ด้วย โดยเมธอด least นี้จะมีเวลาคงที่เพราะใช้คุณสมบัติ random access ของอาร์เรย์โดยตรง สำหรับตัวแปรอื่นที่อาจมีผลต่อเวลาในการทำงานของโค้ดนั้น เมธอด add ที่ sumDigits เรียกใช้นั้นมีเวลาคงที่เพราะการขยายอาร์เรย์นั้นไม่เกิดขึ้น (นอกจากกรณีที่เกิดค่าทดไปตอนท้ายสุด ซึ่งอยู่นอกลูป) สรุปว่าเวลาของเมธอด add ใหม่นี้ก็จะขึ้นอยู่กับขนาดของอาร์เรย์ลิสต์ที่กำหนดไว้ตอนลูปเป็นสำคัญ ซึ่งค่านี้คือ O(จำนวนหลักเลขของตัวที่มีจำนวนหลักมากที่สุด) นั่นเอง

ลิงค์ลิสต์ (LinkedList)

สำหรับลิงค์ลิสต์ในไลบรารี่ของจาวานั้นมีข้อแตกต่างจากลิสต์ที่เรียนในบทก่อนคือ ของจาวาเป็นลิสต์ที่มีพอยต์เตอร์ไปสองทิศทาง นอกจากนี้ ชื่อเมธอดของอิเทอเรเตอร์ก็ไม่เหมือนกัน

เปรียบเทียบลิงค์ลิสต์กับอาร์เรย์ลิสต์

บางคนอาจยังสงสัย จากที่ดูอาร์เรย์ลิสต์มาแล้ว ว่าทำไมจะต้องใช้ลิงค์ลิสต์ด้วยในเมื่ออาร์เรย์ลิสต์ก็ใช้ได้อยู่แล้ว อาร์เรย์ลิสต์นั้นเร็วมาก เพราะว่า

- ไม่ต้องสร้างโนดสำหรับสมาชิกแต่ละตัวในลิสต์ และ
- ยังสามารถเรียกใช้เมธอด System.arraycopy ได้อีกด้วย

แต่ถ้าเราลองมาคิดกรณีเหล่านี้ดู เช่น มีการเติมสมาชิกเข้าที่หัวลิสต์บ่อยๆ หรือมีการใช้อิเทอเรเตอร์ลูปเอาของออกจากบนตัวลิสต์

- เวลาของพวกนี้จะเป็นค่าคงที่ ถ้าเราใช้ลิงค์ลิสต์
- แต่จะเป็น O(n) ถ้าเราใช้อาร์เรย์ลิสต์ เพราะต้องเลื่อนสมาชิกทั้งอาร์เรย์

ดังนั้นในกรณีที่กล่าวไว้ข้างบน การใช้ลิงค์ลิสต์จึงถือได้ว่าดีกว่า แต่อย่าลืมว่า การเข้าถึงสมาชิกโดยใช้ค่าดัชนีนั้น ลิงค์ลิสต์ใช้เวลา O(n) ในขณะที่อาร์เรย์ลิสต์ใช้เวลาคงที่ ดังนั้น ก่อนจะตัดสินใจใช้ลิงค์ลิสต์ในโปรแกรมใด ควรเปรียบเทียบกับอาร์เรย์ลิสต์ก่อนทุกครั้ง

ข้อแตกต่างอีกอย่างหนึ่งของลิงค์ลิสต์กับอาร์เรย์ลิสต์ก็คือ คอนสตรัคเตอร์ของลิงค์ลิสต์นั้นไม่ต้องกำหนดความจุ เพราะยังไงลิงค์ลิสต์ก็ขยายตามความจำเป็นจริงๆของโปรแกรม (ของอาร์เรย์ลิสต์ นั้นมีการขยายน้อยครั้ง ครั้งละหลายช่อง) ลิงค์ลิสต์ยังไม่มี ensureCapacity และ trimToSize ที่ใช้เฉพาะกับอาร์เรย์ลิสต์เท่านั้นอีกด้วย

นอกจากนี้ ลิงค์ลิสต์ยังมีเมธอด 6 เมธอดที่ไม่มีในอาร์เรย์ลิสต์ นั่นคือ

- addFirst(E element)
- getFirst
- removeFirst
- addLast(E element)
- getLast
- removeLast

เวลากรณีแย่สุดของเมธอดพวกนี้มีค่าคงที่ เมธอดเหล่านี้ทำให้ใช้ลิงค์ลิสต์สร้างคิวได้ง่าย รวมไปถึงคิวแบบสองหน้าด้วย แต่เมธอดพวกนี้ไม่มีในอาร์เรย์ลิสต์ ดังนั้นถ้าใช้ลิงค์ลิสต์และเลือกใช้เมธอดพวกนี้ไป เราก็อาจจะยุ่งในการปรับเมธอดในภายหลังเมื่อเปลี่ยนไปใช้อาร์เรย์ลิส (จริงๆแล้วใช้เมธอดของลิสต์ที่มีอยู่แล้วก็ได้ เช่น removeFirst() สามารถใช้ remove(0) แทนได้ แต่ใช้ removeFirst() ก็สะดวกกว่า อ่านง่ายกว่า)

เรามาลองเทียบเมธอดของลิงค์ลิสต์กับอาร์เรย์ลิสต์กันดู

public boolean add(E Element) เป็นเมธอดที่ใช้ใส่ของเข้าท้ายลิสต์

- เวลาที่เมธอดนี้ใช้ในลิงค์ลิสต์นั้นคงที่ เพราะจาวาสามารถหาท้ายลิสต์จากลิงค์ลิสต์สองทางได้เลย
- แต่ถ้าเราใช้อาร์เรย์ลิสต์เรียกเมธอดนี้อาจจะต้องขยายอาร์เรย์ เพราะฉะนั้นเวลาจึงเป็น O(n)

ดังนั้นถ้าเราใส่ของลงท้ายลิสต์ n ครั้ง ลิงค์ลิสต์ก็จะใช้เวลา O(n) เพราะใส่ไป n ครั้ง ส่วนอาร์เรย์ลิสต์นั้น ครั้งแรกต้องนับสมาชิก แต่พอรู้แล้วครั้งต่อๆไปก็ใช้เวลาคงที่ ดังนั้นเวลาจึงเป็น O(n) เหมือนกัน

public E get(int index) เป็นเมธอดที่รีเทิร์นสมาชิกที่ตำแหน่งที่บอกด้วยค่าดัชนี index นั้น

- สำหรับลิงค์ลิสต์นั้น ต้องนับจากหัวลิสต์ไป ดังนั้นเวลาเป็น O(n)
- แต่ถ้าเป็นอาร์เรย์ลิสต์ สามารถใช้ random access ได้เลย เวลาจึงเป็น ค่าคงที่

public E set(int index, E element) เป็นเมธอดที่เปลี่ยนสมาชิกที่ index เป็นตัวใหม่แล้วรีเทิร์นตัวเก่าออกมา

- สำหรับลิงค์ลิสต์นั้น เมธอดนี้ใช้เวลา O(n) เช่นเดียวกับเมธอด get
- ส่วนอาร์เรย์ลิสต์นั้นใช้ เวลาคงที่เช่นเดิม

ถ้าเราเปรียบเทียบเนื้อที่ที่ใช้เก็บสมาชิก สมมติว่าเรามีสมาชิก 1000 ตัว ลิงค์ลิสต์ที่มีสมาชิก 1000 ตัว ก็จะต้องมี 1000 Entry object ต่อกันไป ซึ่งเห็นได้ชัดว่า ลิงค์ลิสต์นั้นกินเนื้อที่สำหรับ

เก็บสมาชิกแต่ละตัวค่อนข้างมาก แต่สำหรับอาร์เรย์ลิสต์นั้นไม่ต้องเสียพื้นที่เก็บของสำหรับสมาชิกแต่ละตัวมากนักเพราะเก็บใส่อาร์เรย์เลย แต่เนื้อที่จะไปเปลืองตรงที่ขนาดอาร์เรย์นั้นต้องทำให้ใหญ่เกินจำนวนสมาชิกจริงๆนั่นเอง

โครงสร้างคลาสของลิงค์ลิสต์

รูป 5.19 แสดงคลาส Entry ซึ่งเป็นคลาสภายในของลิงค์ลิสต์ ซึ่งนี่ก็คือโครงสร้างของโนดหนึ่งนั่นเอง

```
1:    private static class Entry<E>{
2:       E element;
3:       Entry<E>next;
4:       Entry<E>previous;
5:
6:       Entry(E element, Entry<E>next, Entry<E>previous){
7:          this.element =element;
8:          this.next =next;
9:          this.previous =previous;
10:      }
12:   }
```

รูป 5.19 คลาส Entry

คอนสตรัคเตอร์ของลิงค์ลิสต์

รูป 5.20 แสดงโค้ดโครงสร้างภายในของลิงค์ลิสต์อย่างคร่าวๆ โดยแสดงเพียงส่วนหัวของโค้ดกับส่วนคอนสตรัคเตอร์เท่านั้น โนด header นั้นถือเป็นโนดปลอม (dummy node) นั่นเอง ส่วนการสร้าง header ขึ้นมานั้น ตัว Entry แม้ว่าจะถูกเซ็ตให้มีพารามิเตอร์เป็น null ทั้งหมด แต่ค่าพารามิเตอร์นั้นก็จะถูกเปลี่ยนโดยคอนสตรัคเตอร์อีกทีหนึ่ง

เพื่อให้เห็นภาพชัดเจนยิ่งขึ้น เรามาลองสร้างลิสต์กันดู สมมติว่าเราได้รันเมธอด

LinkedList<String> name = new LinkedList<String>();

คอนสตรัคเตอร์จะถูกเรียกใช้ และลิงค์ลิสต์จะมีรูปร่างดังรูป 5.21 โดย next กับ previous จะชี้มาที่ตัว Entry เอง

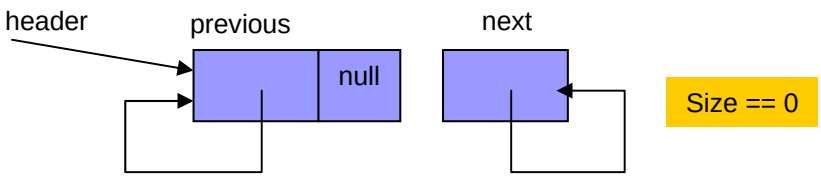
แล้วสมมติว่าเราทำต่อจากนี้โดยเรียกใช้เมธอด

name.add("Rana");

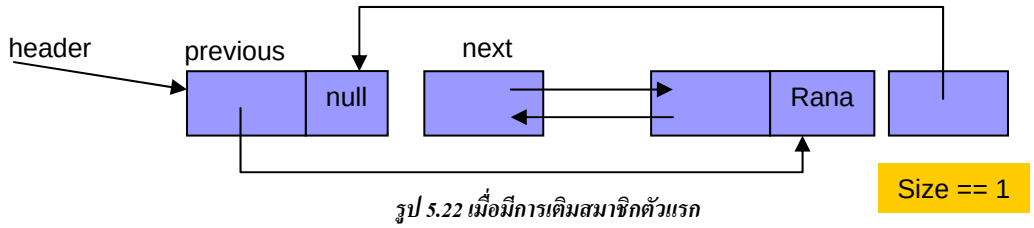
จะเกิดการสร้าง Entry ขึ้นใหม่ กลายเป็นลิงค์ลิสต์ที่ต่อเป็นวงกลม (circular linked list) ดังรูป 5.22

```
1:    public class LinkedList<E>extends
AbstractSequentialList<E>, implements List<E>,
java.lang.Cloneable, java.io.Serializable
2:    {
3:      private transient int size =0; //จำนวนสมาชิกในลิสต์
4:      private transient Entry<E>header =new
Entry<E>(null,null,null);
6:
7:      public LinkedList() {
8:
9:         header.next =header.previous =header;
10:     }
12:     public LinkedList(Collection c) {
14:        this();
15:        addAll(c);
16:     }
17:      ... //โค้ดส่วนอื่นๆ
18:  }
```

รูป 5.20 โครงสร้างโดยคร่าวๆของลิงค์ลิสต์ในจาวา



รูป 5.21 เมื่อคอนสตรัคเตอร์ถูกเรียกใช้



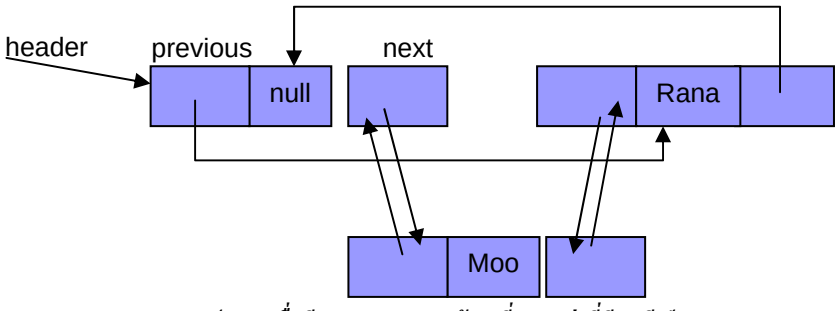
รูป 5.22 เมื่อมีการเติมสมาชิกตัวแรก

เมธอดต่างๆของลิงค์ลิสต์

ถ้าเราทำต่อจากรูป 5.22 ด้วยการเรียกเมธอด

name.add(0, “Moo”);

จะเป็นการเอาสตริง Moo มาแทรกที่ตำแหน่งแรกที่ถัดจากโนดปลอม ไม่เสียเวลาเลื่อนสมาชิกตัวอื่นๆไปท้ายลิสต์แบบตอนที่ใช้อาร์เรย์ ดังที่ได้กล่าวไปแล้ว แต่ยังใช้การลูปตอนหาตำแหน่งที่ 0 (การหาตำแหน่งสิ้นสุดลงในลูปแรก เวลาจึงไม่เป็น O(n)) รูป 5.23 แสดงสถานะของลิสต์เมื่อมีการแทรก Moo เข้าไปดังกล่าว



รูป 5.23 เมื่อมีการแทรก Moo เข้ามาที่ตำแหน่งที่มีดัชนีเป็น 0

เมธอด add(int index, E element) นั้น ลูปหาตำแหน่ง index ก่อน โดยเริ่มลูปจากหัวลิสต์ถ้า index < size/2 ไม่เช่นนั้นจะเริ่มลูปมาจากท้ายลิสต์ โดยการลูปนั้นจะใช้ เมธอดที่ชื่อ entry (รูป 5.24) หลังจากหาตำแหน่งเจอแล้วก็จะจัดการสร้างโนดใหม่และจัดพอยต์เตอร์ โดยการจัดพอยต์เตอร์นั้นทำโดย private method ที่ชื่อ addBefore (รูป 5.25)

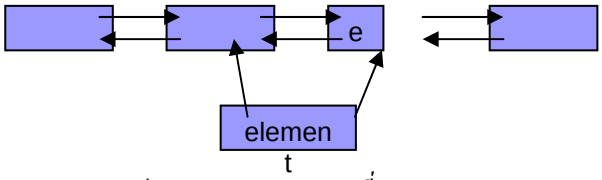
เมธอด entry นั้นเข้าใจได้ไม่ยากนัก ดังนั้นจึงจะอธิบายแต่เมธอด addBefore เท่านั้น โดยเมธอด addBefore นั้นใส่ เอนทรี่ออบเจ็กต์ ซึ่งมี element อยู่ เข้าไปข้างหน้าของ e (ซึ่งเป็นเอนทรี่ออบเจ็กต์ด้วย) แล้วรีเทิร์นผลเป็น เอนทรี่ออบเจ็กต์ตัวที่เราใส่ไปในลิสต์นี้ ซึ่งมี element อยู่ภายในนั่นเอง รูป 5.26 แสดงสภาวะของลิงค์ลิสต์เมื่อโค้ดบรรทัดที่ 4 ของ addBefore ทำงาน ส่วนรูป 5.27 แสดงสภาวะของลิงค์ลิสต์หลังจากโค้ดบรรทัดที่ 7 และ 8 ของ addBefore ทำงาน โค้ดของเมธอด add เองนั้นอยู่ในรูป 5.28

```
1:    private Entry entry(int index) {
2:      if (index < 0 || index >= size)
3:       throw new IndexOutOfBoundsException("Index: "+index+
4:                 ", Size: "+size);
5:      Entry e = header;
6:      if (index < (size >> 1)) {
7:       for (int i = 0; i <= index; i++)
8:        e = e.next;
9:      } else {
10:      for (int i = size; i > index; i--)
11:       e = e.previous;
12:     }
13:     return e;}
```

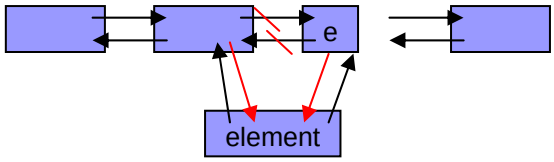
รูป 5.24 เมธอด entry

```
1:    private Entry<E> addBefore(E element, Entry<E> e)
2:
3:    { //ก่อนอื่นใส่ newEntry ข้างหน้า e (พอยต์เตอร์ยังจัดไม่ครบ)
4:       Entry<E> newEntry = new Entry<E>(element, e, e.previous);
5:
6:      //จัดพอยต์เตอร์ซะ
7:      newEntry.previous.next = newEntry;
8:      newEntry.next.previous = newEntry;
9:
10:     size++;
11:     modCount++;
12:     return newEntry;
13:
14:   }
```

รูป 5.25 โค้ดของ addBefore



รูป 5.26 สภาวะตอนบรรทัดที่ 4 ของ addBefore



รูป 5.27 สภาวะหลังบรรทัด 7 และ 8 ของ addBefore ถูกทำงาน

```
1:    public void add(int index, E element){
2:      if(index == size)
3:        addBefore(element, header);
4:      else
5:        addBefore(element, entry(index));
6:    }
```

รูป 5.28 โค้ดของเมธอด add ที่ใช้ค่าดัชนีด้วย

โค้ดของเมธอดที่น่าสนใจ

ส่วนโค้ดของเมธอดอื่นๆจากจาวา 1.4 นั้นถูกรวมไว้ในรูป 5.29 ถึง 5.37 เพื่อไว้ใช้อ้างอิง

```
1:    public Object getFirst() {
2:     if (size==0)
3:       throw new NoSuchElementException();
4:     return header.next.element;
5:    }
6:
7:    public Object getLast() {
8:     if (size==0)
9:       throw new NoSuchElementException();
10:    return header.previous.element;
11:   }
```

รูป 5.29 โค้ดของเมธอด getFirst และ getLast

```
1:    public Object removeFirst() {
2:     Object first = header.next.element;
3:     remove(header.next);
4:     return first;
5:     }
6:
7:     public Object removeLast() {
8:     Object last = header.previous.element;
9:     remove(header.previous);
10:    return last;
11:    }
```

รูป 5.30 โค้ดของเมธอด removeFirst และ removeLast

```
1:    public void addFirst(Object o) {
2:       addBefore(o, header.next);
3:     }
4:
5:     public void addLast(Object o) {
6:       addBefore(o, header);
7:     }
8:
9:     public boolean contains(Object o) {
10:      return indexOf(o) != -1;
11:    }
```

รูป 5.31 โค้ดของเมธอด addFirst addLast และ contains

```
1:    public int size() {
2:     return size;
3:    }
4:
5:    public boolean add(Object o) {
6:     addBefore(o, header);
7:     return true;
8:    }
9:
10:   public boolean addAll(Collection c) {
11:    return addAll(size, c);
12:   }
```

รูป 5.32 โค้ดของเมธอด size add(Object o) และ addAll

```
1:    public boolean remove(Object o){//o คือตัวที่จะเอาออก
2:      if (o==null){
3:       for (Entry e =header.next; e !=header; e =e.next){
4:        if (e.element==null){
5:         remove(e);
6:         return true;
7:        }
8:       }
9:      } else {
10:      for (Entry e =header.next; e !=header; e =e.next){
11:       if (o.equals(e.element)){
12:        remove(e);
13:        return true;
14:       }
15:      }
16:     }
17:     return false;}
```

รูป 5.33 โค้ดของเมธอด remove

```
1:    public boolean addAll(int index, Collection c){
2:     int numNew =c.size();
3:     if (numNew==0)
4:       return false;
5:     modCount++;
6:
7:     Entry successor =(index==size ? header :entry(index));
8:     Entry predecessor =successor.previous;
9:     Iterator it =c.iterator();
10:    for (int i=0; i<numNew; i++){
11:      Entry e =new Entry(it.next(), successor,
predecessor);
12:      predecessor.next =e;
13:      predecessor =e;
14:    }
15:    successor.previous =predecessor;
16:
17:    size +=numNew;
18:    return true;
19:   }
```

รูป 5.34 โค้ดของเมธอด addAll ที่ใช้ค่าดัชนีในการกำหนดจุดเริ่มใส่

```
1:    public void clear() {
2:     modCount++;
3:     header.next = header.previous = header;
4:     size = 0;}
6:
7:    public Object get(int index) {
8:     return entry(index).element;
9:    }
10:
11:   public Object set(int index, Object element) {
12:     Entry e = entry(index);
13:     Object oldVal = e.element;
14:     e.element = element;
15:     return oldVal;
16:   }
```

รูป 5.35 โค้ดของเมธอด clear get และ set

อิเทอเรเตอร์ของลิงค์ลิสต์ในจาวา

ชื่อคลาสจริงๆนั้นคือ ListItr ซึ่งอิมพลีเม้นท์ ListIterator interface ListItr นั้นเป็น private class ภายในคลาส LinkedList ดังนั้นคลาสนี้จึงสร้างออบเจ็กต์ของตัวมันเองโดยตรงไม่ได้ ต้องสร้างจากการเรียกเมธอดของลิงค์ลิสต์เท่านั้น

ลิงค์ลิสต์นั้นสามารถเรียกใช้เมธอดเพื่อสร้างอิเทอเรเตอร์ ListItr ได้สองเมธอด

เมธอดแรกคือ public ListIterator<E> listIterator() ซึ่งรีเทิร์น ListIterator ออบเจ็กต์ซึ่งชี้ไปที่หัวลิสต์ เราสามารถเรียกใช้ได้ดังตัวอย่างต่อไปนี้

ListIterator<String> itr1 = animal.listIterator();

ส่วนอีกเมธอดหนึ่งคือ public ListIterator<E> listIterator(final int index) ซึ่งรีเทิร์น ListIterator ซึ่งชี้ไปยังตำแหน่งที่บอกด้วยค่า index เมธอดนี้ใช้เวลา O(n) เราสามารถเรียกใช้ได้ดังนี้

ListIterator<String> itr2 = animals.listIterator(3);

```
1:    public Object remove(int index) {
2:      Entry e = entry(index);
3:      remove(e);
4:      return e.element;
5:    }
6:
7:    public int indexOf(Object o) {
8:      int index = 0;
9:      if (o==null) {
10:       for (Entry e = header.next; e != header; e = e.next) {
11:         if (e.element==null)
12:           return index;
13:         index++;
14:       }
15:     } else {
16:       for (Entry e = header.next; e != header; e = e.next) {
17:         if (o.equals(e.element))
18:           return index;
19:         index++;
20:       }
21:     }
22:     return -1;
23:   }
```

รูป 5.36 โค้ดของเมธอด remove และ indexOf

โค้ดคอนสตรัคเตอร์ของ ListItr และการเรียกใช้ (jdk1.4) นั้นอยู่ในรูป 5.38

ในการใช้ลูปเพื่อลูปไปกับตัวลิสต์อิเทอเรเตอร์นั้นสามารถใช้ลูปแบบพิเศษได้

```
for(String s : animals)
    System.out.println(s);
```

ซึ่งหมายถึง สำหรับ s แต่ละตัว ให้พิมพ์ค่าของมันออกไป

ฟิลด์ของคลาส ListItr นั้นแสดงในรูป 5.39 ตัวเมธอดของ ListItr ก็เหมือนกับของ ListIterator เพราะว่าอิมพลีเม้นท์อินเตอร์เฟสมา แต่เราควรมาดูรายละเอียดกันซักนิดหนึ่งเพื่อกันความสับสน

```
1:    public int lastIndexOf(Object o) {
2:      int index = size;
3:      if (o==null) {
4:       for (Entry e = header.previous; e != header; e =
e.previous) {
5:        index--;
6:        if (e.element==null)
7:         return index;
8:       }
9:      } else {
10:      for (Entry e = header.previous; e != header; e =
e.previous) {
11:       index--;
12:       if (o.equals(e.element))
13:        return index;
14:      }
15:     }
16:     return -1;
17:   }
```

รูป 5.37 โค้ดของเมธอด lastIndexOf

```
1:    public ListIterator listIterator(int index) {
2:     return new ListItr(index);
3:    }
4:
5:    ListItr(int index) {
6:      if (index < 0 || index > size)
7:        throw new IndexOutOfBoundsException("Index:
"+index+", Size: "+size);
8:      if (index < (size >> 1)) {
9:        next = header.next;
10:       for (nextIndex=0; nextIndex<index; nextIndex++)
11:           next = next.next;
12:     } else {
13:       next = header;
14:       for (nextIndex=size; nextIndex>index;
nextIndex--)
15:           next = next.previous;
16:     }
17:   }
```

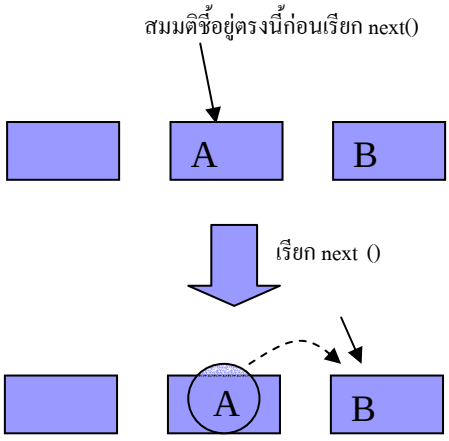
รูป 5.38 คอนสตรัคเตอร์ของ ListItr และการเรียกใช้จากคอนสตรัคเตอร์ของ listIterator (jdk 1.4)

```
1:    private class ListItr implements ListIterator {
2:     private Entry lastReturned = header;
3:     private Entry next;
4:     private int nextIndex;
5:     private int expectedModCount = modCount;
6:
7:      ... เมธอดต่างๆ
8:
9:    }
```

รูป 5.39 ฟิลด์ต่างๆของคลาส ListItr

เมธอดแรกคือ public E next()

การทำงานของเมธอดนี้นั้นแสดงในรูป 5.40 ถ้าแต่เดิมนั้นอิเทอเรเตอร์ชี้ไปยังโนดที่มี A เป็นสมาชิกแล้วละก็ เมธอด next จะรีเทิร์น A เป็นคำตอบ แล้วจะเลื่อนไปชี้ยังโนดต่อไป จึงจะสิ้นสุดการทำงาน

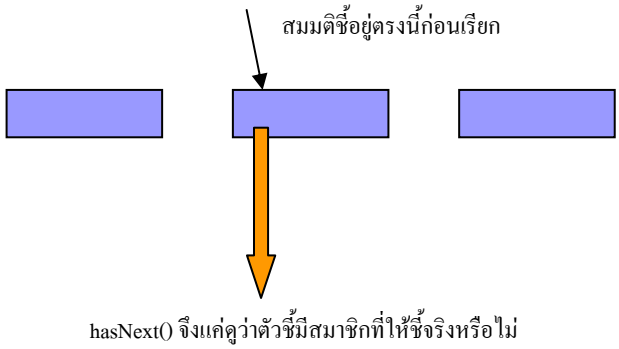


1. จะเอาของตรงที่ชี้อยู่เดิม (A) นั้นเป็นคำตอบ
2. แล้วเลื่อนตำแหน่งตัวชี้ไปยังโนดถัดไป

รูป 5.40 การทำงานของ next() จากอิเทอเรเตอร์ของลิงค์ลิสต์

เมธอดที่สองคือ public boolean hasNext() (แสดงในรูป 5.41)

เมธอดนี้จะตรวจว่าตำแหน่งที่อิเทอเรเตอร์ชี้อยู่มีสมาชิกของลิสต์จริงหรือไม่ ไม่มีการเลื่อนพอยต์เตอร์ที่อิเทอเรเตอร์ชี้ โค้ดของเมธอด hasNext กับเมธอด next นั้นอยู่ในรูป 5.42



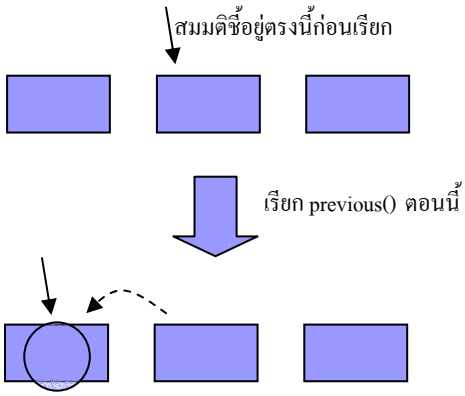
รูป 5.41 การทำงานของ hasNext()

```
1:    public boolean hasNext() {
2:          return nextIndex != size;
3:      }
4:
5:    public Object next() {
6:          checkForComodification();
7:          if (nextIndex == size)
8:            throw new NoSuchElementException();
9:
10:         lastReturned = next;
11:         next = next.next;
12:         nextIndex++;
13:         return lastReturned.element;
14:   }
```

รูป 5.42 โค้ดของ hasNext() กับ next()

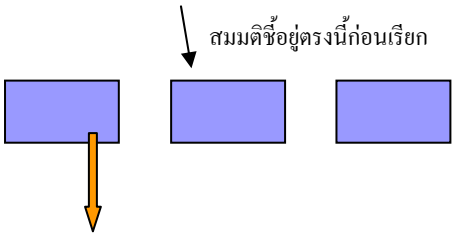
เมธอดต่อไปคือ public E previous() ซึ่งแสดงในรูป 5.43
เมธอดนี้จะทำการเลื่อนตำแหน่งที่อิเทอเรเตอร์ชี้ไปทางซ้ายของลิสต์หนึ่งตำแหน่งก่อน จากนั้นจึงรีเทิร์นสมาชิกของลิสต์ที่อยู่ที่ตำแหน่งใหม่นั้นเป็นคำตอบของเมธอด

เมธอดที่คู่กับ previous() ก็คือ public boolean hasPrevious() ซึ่งแสดงในรูป 5.44
โดยการทำงานของ hasPrevious() ก็คือตรวจสอบว่า previous() มีคำตอบให้รีเทิร์นได้จริงหรือไม่



1. เลื่อนตำแหน่งตัวชี้ไปยังโนดก่อนหน้า
2. จะเอาของตรงที่ชี้ใหม่นั้นเป็นคำตอบ

รูป 5.43 การทำงานของ previous() จากลิสต์อิเทอเรเตอร์



hasPrevious() จึงดูว่า ตอนที่เรียก previous นั้น
มีค่าที่เป็นคำตอบของ previous จริงหรือไม่

รูป 5.44 การทำงานของ hasPrevious() จากลิสต์อิเทอเรเตอร์

โค้ดของ previous() และ hasPrevious() นั้นแสดงในรูป 5.45

เพื่อความเข้าในที่ดีขึ้น เรามาลองดูตัวอย่างการใช้ previous() และ hasPrevious() กัน

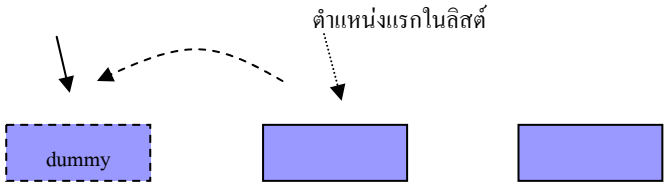
```
1:    public boolean hasPrevious() {
2:       return nextIndex != 0;
3:    }
4:
5:    public Object previous() {
6:       if (nextIndex == 0)
7:          throw new NoSuchElementException();
8:       lastReturned = next = next.previous;
9:       nextIndex--;
10:      checkForComodification();
11:      return lastReturned.element;
12:   }
```

รูป 5.45 โค้ดของ previous() กับ hasPrevious()

ตัวอย่างที่ 5-7

ถ้าเริ่มใช้ previous ที่หัวลิสต์ จะเกิดอะไรขึ้น

ถ้าอิเทอเรเตอร์อยู่ตรงตำแหน่งแรก จะเรียกใช้ previous() ไม่ได้ เพราะจะได้คำตอบเป็นดัมมี่โนด ซึ่งเราไม่ต้องการ ดังแสดงในรูป 5.46 โค้ดในรูป 5.45 จึง throw Exception



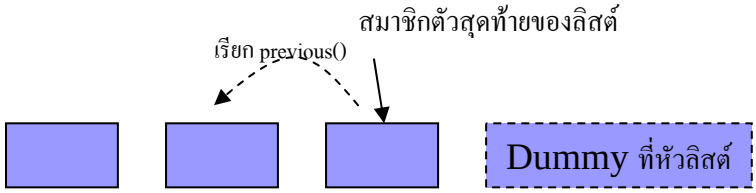
รูป 5.46 previous() ที่ตำแหน่งแรก ทำให้เลื่อนกลับไปชี้ดัมมี่โนด

ตัวอย่างที่ 5-8

ถ้าเริ่มใช้ previous() ลูปจากท้ายลิสต์ เราจะได้สมาชิกตัวสุดท้ายในลิสต์ได้อย่างไร

ถ้าในตอนแรกอิเทอเรเตอร์ชี้ที่โนดสุดท้ายที่เก็บสมาชิกของลิสต์ ดังรูป 5.47 การเรียก previous() จะทำให้อิเทอเรเตอร์เลื่อนไปยังตำแหน่งรองสุดท้ายในลิสต์ก่อนที่จะรีเทิร์นคำตอบ

ดังนั้น previous() จึงไม่มีวันที่จะรีเทิร์นสมาชิกตัวสุดท้ายในลิสต์ได้ (แต่สมาชิกตัวสุดท้ายในลิสต์นี้ยังคงหาได้ด้วย next())



รูป 5.47 การเรียก previous() เมื่ออิเทอเรเตอร์ชี้ไปยังตำแหน่งสุดท้ายของลิสต์

ถ้าเราอยากให้ previous() รีเทิร์นสมาชิกตัวสุดท้ายในลิสต์ เราจะต้องให้อิเทอเรเตอร์ชี้ไปยังตำแหน่งที่เลยตำแหน่งสุดท้ายไปเสียก่อน ซึ่งสามารถทำได้ในจาวา เพราะลิงค์ลิสต์ของจาวานั้นเป็นลิงค์ลิสต์แบบต่อเป็นวงกลม ตำแหน่งสุดท้ายของลิสต์นั้น โนดของมันจะต่อกับโนดปลอมที่หัวลิสต์ ฉะนั้นถ้าเราเลื่อนอิเทอเรเตอร์ไปชี้ที่โนดปลอม ตอนเรียก previous() อิเทอเรเตอร์ก็จะเลื่อนตำแหน่งที่ชี้มาเป็นตำแหน่งสุดท้ายของลิสต์นั่นเอง โค้ดในรูป 5.48 แสดงการเริ่มลูปจากท้ายลิสต์โดยใช้หลักการแก้ปัญหานี้

```
1:    ListIterator itr =
2:                 animals.listIterator(animals.size());
3:    //iterator now at dummy node
4:    while (itr.hasPrevious( ))
5:           itr.previous( );
```

รูป 5.48 การเริ่มลูปจากท้ายลิสต์ที่รีเทิร์นสมาชิกตัวสุดท้ายได้ถูกต้อง

ยังมีเมธอดของอิเทอเรเตอร์อีกบางเมธอดที่น่าสนใจ ซึ่งจะได้นำเสนอต่อไปนี้

public void add(E element)

- เมธอดนี้ทำการใส่สมาชิกใหม่เข้าไปข้างหน้าสมาชิกที่จะถูกรีเทิร์นด้วย next() (ถ้ามี)
- สมาชิกใหม่นี้ต้องไปอยู่ข้างหลังสมาชิกที่จะถูกรีเทิร์นด้วย previous() ด้วย (ถ้ามี)
- ถ้าลิสต์ว่างอยู่ก่อนแล้ว ตัวสมาชิกที่เราใส่นี้ก็จะเป็นสมาชิกตัวแรกของลิสต์นั่นเอง
- เมธอดนี้จะลบล้างบันทึกของอิเทอเรเตอร์ ที่บันทึกว่าพึ่งรีเทิร์นอะไรจาก next() หรือ previous() ไป

เราจะไม่นำเสนอโค้ด แต่จะมีตัวอย่างการใช้งานเมธอดนี้ให้ดังนี้

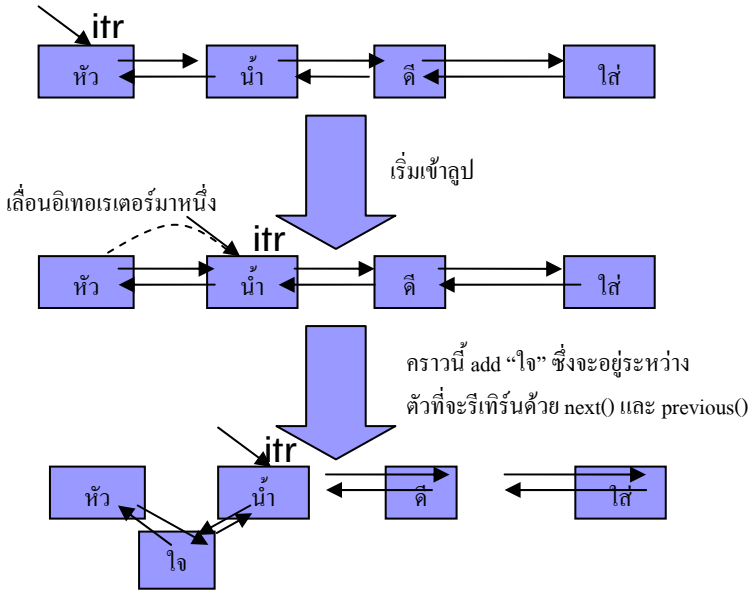
ตัวอย่างที่ 5-9

สมมติว่าเรามีลิสต์ของสตริง “หัว” “น้ำ” “ดี” “ใส่” ต้องการจะใส่คำว่า “ใจ” ลงที่ระหว่างคำแต่ละคำ จะต้องทำอย่างไร

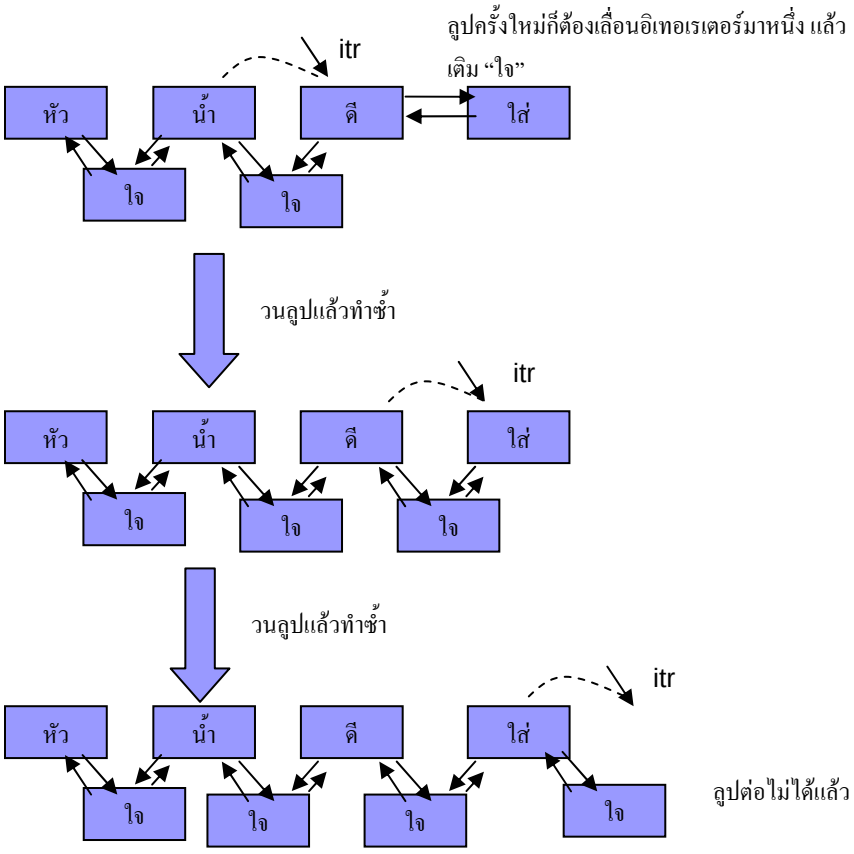
โค้ดที่แก้ปัญหานี้เป็นดังรูป 5.49 โดยมีการทำงานดังรูป 5.50 และ 5.51

```
1:    ListIterator<String> itr  = mylist.listIterator();
2:    while(itr.hasNext()){
3:       itr.next();
4:       itr.add(“ใจ”);
5:    }
```

รูป 5.49 แทรก “ใจ” ไประหว่างทุกคำ



รูป 5.50 การทำงานของโค้ดในรูป 5.49 (ช่วงเริ่มต้น)



รูป 5.51 การทำงานของโค้ดในรูป 5.49 (ต่อจากรูป 5.50)

เมธอดที่น่าสนใจอีกเมธอดหนึ่งของลิสต์อิเทอเรเตอร์ก็คือ

public void remove()

- เอาสมาชิกตัวที่พึ่งเป็นผลลัพธ์ของ next() หรือ previous() ออกจากลิสต์
- เมธอดนี้ใช้สลับกับ next() หรือ previous() ได้ครั้งต่อครั้งเท่านั้น
- จะต้องไม่มีการเรียกเมธอด ListIterator.add() ระหว่างการเรียก next() หรือ previous() กับเมธอดนี้

ตัวอย่างที่ 5-10

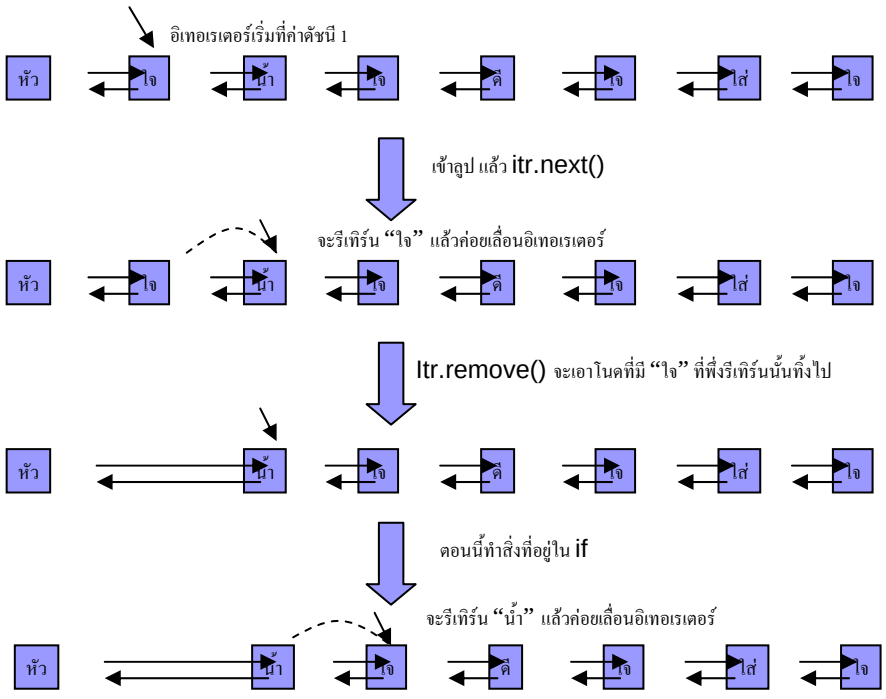
สมมติว่าเราต้องการเอา สมาชิกที่มีดัชนีเป็นเลขคี่ออกไปจากผลลัพธ์ของตัวอย่างที่แล้วให้หมด

จะต้องทำอย่างไร โค้ดนั้นอยู่ในรูปที่ 5.52

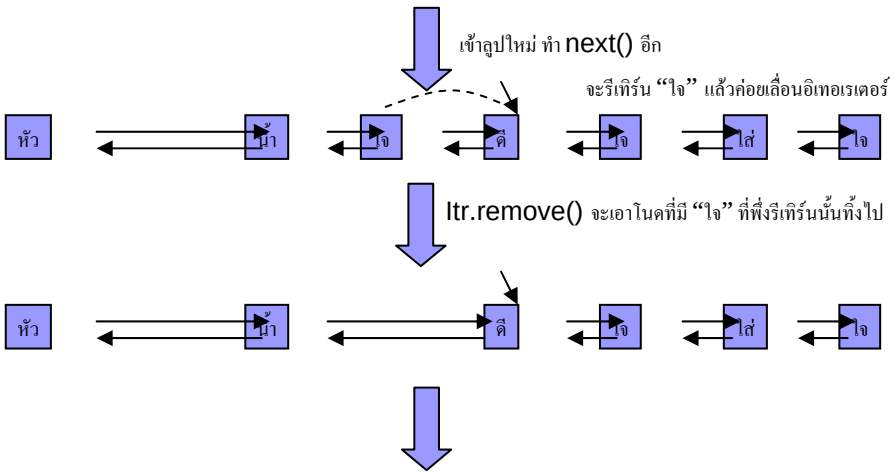
```
1:    ListIterator<String> itr = mylist.listIterator(1);
2:    while(itr.hasNext()){
3:       itr.next();
4:       itr.remove();
5:       if(itr.hasNext())
6:          itr.next();
7:    }
```

รูป 5.52 การเอาสมาชิกที่มีดัชนีเป็นเลขคี่ทิ้งไป

เรามาดูภาพตัวอย่างกัน (รูป 5.53)



(รูปนี้มีต่อหน้าถัดไป)



พอเสร็จแล้วจะเห็นว่า "ใจ" ถูกเอาออกไปหมด

รูป 5.53 ภาพการทำงานของโค้ดในรูป 5.52 สังเกตเมธอด remove

เดี๋ยวเรามาดูเมธอดของอิเทอเรเตอร์กันต่อ คราวนี้คือเมธอด set

public void **set**(E o)

- เอา o แทนที่สมาชิกตัวท้ายสุดที่พึ่งถูกรีเทิร์นเป็นคำตอบของ next() หรือ previous()
- นอกจากนี้จะต้องไม่มีการเรียกเมธอด ListIterator.add() และ ListIterator.remove() ระหว่างการเรียก next() หรือ previous() กับเมธอดนี้

ตัวอย่างที่ 5-11

ถ้าเราต้องการเปลี่ยนคำว่า "ใจ" ของลิสต์ในตัวอย่างที่แล้ว ให้เป็นคำว่า "จาย" จะต้องเขียนโค้ดอย่างไร

โค้ดที่เป็นคำตอบของตัวอย่างนี้อยู่ในรูป 5.54 (ตัวอย่างนี้ผมจะไม่วาดรูปสิ่งที่เกิดขึ้น ให้นักเรียนลองไปวาดดูเป็นแบบฝึกหัด)

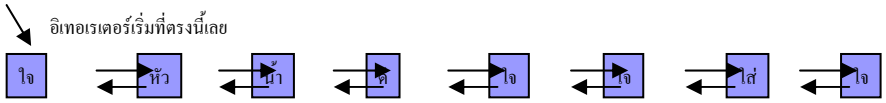
```
1:     ListIterator<String> itr = mylist.listIterator();
2:     while(itr.hasNext()){
3:        if(itr.next().equals("ใจ"))
4:           itr.set("จาย");
5:     }
```

รูป 5.54 ตัวอย่างการใช้งานเมธอด set ของลิงค์ลิสต์อิเทอเรเตอร์

สำหรับเรื่องของลิงค์ลิสต์ที่ผมคิดว่าทุกคนควรจะรู้ ก็จะมีเพียงเท่านี้ ที่เหลือขอให้นิสิตฝึกทำโจทย์ ดัดแปลงโจทย์จากบทก่อนๆมาลองทำโดยใช้ไลบรารี่ของจาวานี้ดู จะได้สามารถใช้งานจาวาได้อย่างมีประสิทธิภาพมากขึ้น

แบบฝึกหัด

1. ถ้าเราต้องการเอาสมาชิกที่มีคำว่า "ใจ" ที่อยู่ตรงไหนก็ได้ ออกจากลิงค์ลิสต์ จะต้องเขียนโค้ดอย่างไร ให้ตัวอย่างสภาพแรกเริ่มของลิสต์เป็นแบบนี้ (โค้ดที่ใช้ต้องใช้ได้กับการเรียงคำว่า "ใจ" ทุกรูปแบบ)

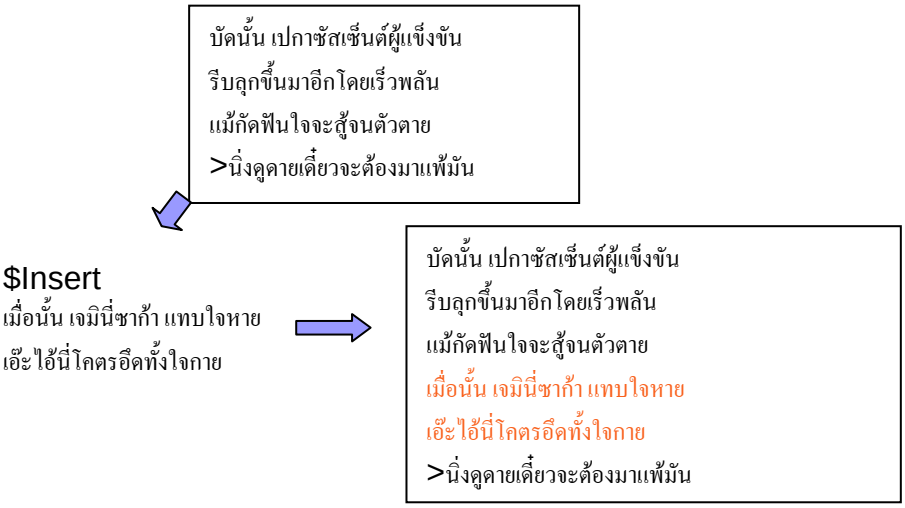


2. สมมติว่ามีอาร์เรย์ลิสต์ที่เก็บข้อมูลทำนองเดียวกับลิงค์ลิสต์ในข้อ 1 ถามว่า ถ้าเราจะทำแบบข้อ 1 กับอาร์เรย์ลิสต์นี้ จะต้องเขียนโค้ดอย่างไร
3. จงวาดรูปสิ่งที่โค้ดในตัวอย่าง 5-11 ทำ อธิบายขั้นตอนของสิ่งที่เกิดขึ้นให้ชัดเจน
4. ถ้าเราออกแบบ line editor (text editor แบบเก่าที่สามารถจัดข้อความได้ทีละบรรทัดเท่านั้น) แบบนี้
   - จัดการกับข้อความได้ทีละบรรทัด (เดี๋ยวนี้พวกโปรแกรมต่างๆให้เราจัดการบรรทัดที่เราต้องการได้ทันที)
   - สมมติว่าแต่ละบรรทัดมีตัวอักษรได้ 75 ตัว

- ให้บรรทัดแรกถือเป็นบรรทัดที่ 0
- มีตัวบอกบรรทัดปัจจุบัน
- ตัวคำสั่งในการทำงานแต่ละตัวเริ่มด้วยเครื่องหมาย $ และจะมีแต่ชุดคำสั่งเท่านั้นที่จะเริ่มต้นด้วยเครื่องหมายนี้

ให้มีคำสั่งที่เราใช้งานได้ สามคำสั่ง คำสั่งแรกคือ Insert ทำงานโดย

- แทรกข้อความเข้าไปในบรรทัดก่อนที่จะถึง current line (ตัวอย่างหน้าจอดังรูปข้างล่าง)

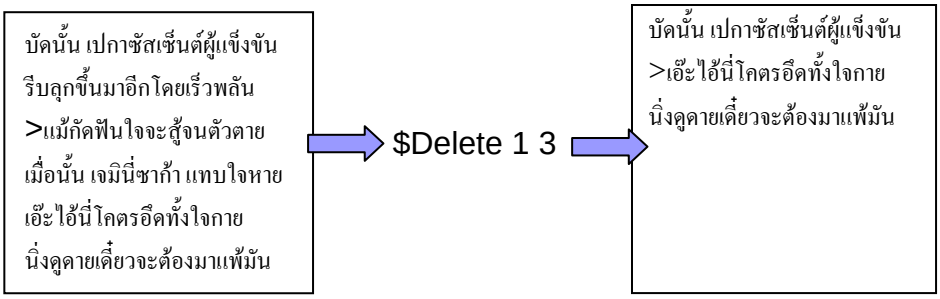


คำสั่งที่สองคือ Delete m n ซึ่งทำงานดังนี้

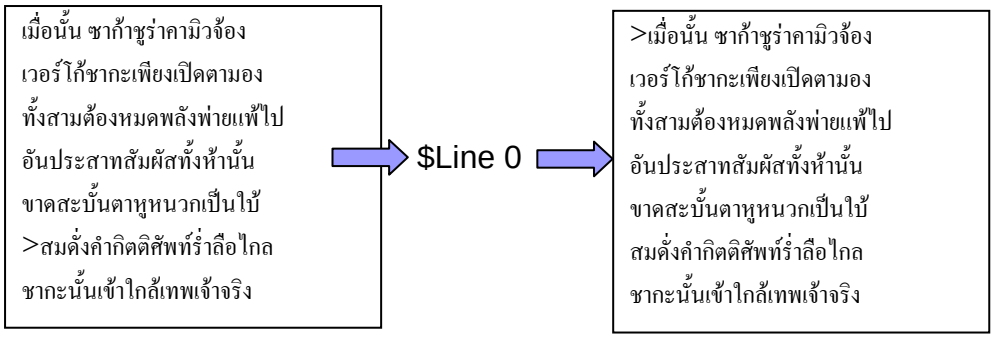
- ลบบรรทัดตั้งแต่บรรทัดที่ m ถึงบรรทัดที่ n (รวมบรรทัดที่ m และ n ด้วย)
- Current line จะย้ายมายังบรรทัดถัดจากบรรทัดสุดท้ายที่ลบไป
  - ยังไง current line ก็ต้องเปลี่ยนตามนี้ไม่ว่าจะอยู่ไหนมาก่อนก็ตาม
  - ถ้าลบบรรทัดสุดท้ายไปด้วย current line ก็จะเลยบรรทัดสุดท้ายไป
- มี error message ในกรณีต่อไปนี้
  - m > n

- m < 0
- n > ตัวเลขของบรรทัดสุดท้าย
- m และ n ต้องเป็นจำนวนเต็มเท่านั้น

ตัวอย่างการทำงานนั้นเป็นดังรูปด้านล่างนี้



คำสั่งที่สามคือ Line m ซึ่ง ทำให้บรรทัดที่ m กลายเป็น current line ดังตัวอย่างข้างล่างนี้



จงออกแบบโครงสร้างข้อมูลเพื่อใช้เก็บข้อความ และการทำงานของเมธอดต่างๆของ line editor ตัวนี้

5. สมมติว่าเรามีลิงค์ลิสต์สองตัว แต่ละตัวมีสมาชิกไม่ซ้ำภายใน จงเขียนโค้ดเพื่อยูเนี่ยนลิงค์ลิสต์สองลิงค์ลิสต์เข้าด้วยกัน ของที่มีซ้ำกันทั้งสองลิสต์ให้ตัดทิ้งไป
6. ถ้าเราต้องการเอาส่วนของลิงค์ลิสต์ตั้งแต่สมาชิกที่ถูกชี้ด้วยอิเทอเรเตอร์ p1 ไปจนถึงสมาชิกที่ถูกชี้ด้วยอิเทอเรเตอร์ p2 ไปใส่ไว้ข้างหน้าสมาชิกที่ถูกชี้ด้วยอิเทอเรเตอร์ p3 เราจะต้องเขียนโค้ดอย่างไร
7. จงเขียนเมธอดเพื่อทำการจัดเรียงข้อมูลแบบ bubble sort ในลิงค์ลิสต์
8. จงเขียนเมธอดเพื่อทำการจัดเรียงข้อมูลแบบ selection sort ได้จากทั้งลิงค์ลิสต์และอาร์เรย์ลิสต์
9. จงเขียนเมธอดสำหรับลิงค์ลิสต์ เพื่อทำ merge sort โดยห้ามสร้างอาร์เรย์ใดๆทั้งสิ้น
10. จงเขียนโค้ดของคิวขึ้นมาโดยใช้ลิงค์ลิสต์ ให้สมาชิกของคิวนี้มีค่าความสำคัญ ในตอน enqueue สมาชิกตัวที่สำคัญกว่าจะได้ไปอยู่หัวคิวเลย