集合與陣列的比較

	集合	陣列
空間大小	不必指定集合大小。無論將元素存入或移除，集合都會動態調整空間以符合需求。	必須指定陣列大小，之後無法改變。元素存取時，不可超過索引上限。
資料類型	可存放物件，放入的物件會自動轉型成 Object 類型，搭配泛型功能就可限制元素的資料類型。	可存放物件或基本類型，但必須符合陣列的資料類型。
資料存取	要將元素取出，可使用： 1. toString 2. for-each。 3. Iterator 物件功能。	要將元素取出，可使用： 1. for 迴圈並搭配索引。 2. for-each。

集合的定義

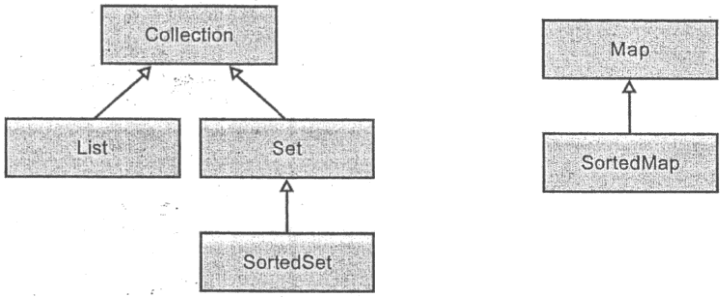
1. The collection framework 是一些 interfaces 及 classes 所組成(定義於 lava.util package 內)，提供各種不同 data structure (資料結構)來儲存、擷取物件。我們可以把 collection 想成各種的黑箱子。箱子是來裝物件的
2. collection 本身是個 object，他是用來裝別的 objects。注意:collection 不可以裝 primitives，primitive type value 必須先用對應的 wrapper class 包成一個 wrapped object 後，才可存進 collection 內
3. 最簡單的 data structure 是 array，array 最大的限制是它不能 resize 再者，它所提供的存取資料的方法太過簡單，不夠強大。而 collection 可解除 array 的限制。注意 array 的 elements 可以是 primitive type 亦可以是 reference type 但 collection 只可是 reference 的型態

集合的架構➔是否順序，是否重複，是否排序 ？

一個 collection 因它所 implements 的 interface 不同，而有不同的特性。

collectio API 四種 interfaces➔ Collection、List、Set 及 Map

繼承關係如下圖:



Collection➔把 Collection 想成是一個袋子，這個袋子可以用來裝許多 objects ，這些 objects 在袋子裡是 non-orders (沒有順序性)，repetition allowed(允許重複)

boolean add(Object o)	新增一個elements到this
boolean addAll(Collection c)	將"c"的所有elements都加到this
boolean remove(Object o)	從this移除一個"o"
boolean removeAll(Collection c)	從this，移除所有"c也有的elements"
void clear()	從this移除所有的elements
int size(Object o)	傳回this的elements個數
boolean isEmpty()	若this是empty，則傳回"true"
boolean contains(Object o)	若this內包含"o"，則傳回"true"
Iterator iterator()	傳回 this的Iterator object
Object[] toArray()	傳回一個 Object array，這個 array 容納了 this 的所有 elements

Set

1. Set 繼承了 Collection 並沒有再宣告新的 method，但對 Collecion 的一些 method 重新給予定義
2. 特性是 non-orders (沒有順序性)，unique(不允許重複)

boolean add(Object o)	若this內不含"o"，才把"o"新增到this
boolean addAll(Collection c)	將"c"的所有elements都加到this，但"this"已有的element，不會再新增到this

List

1. List 也繼承 Collection，List 宣告了一些新的 method，也對 Collection 的一些方法重新給予定義
2. 特性是 orders (有順序性)，repetition allowed(允許重複)

boolean add(Object o)	附加"o"在this的後面
boolean add(Collection c)	將"c"的所有elements都附加到this
void add(int i, Object o)	在position "i"插入"o"
boolean add(int i, Collection c)	在position "i"插入"c"的所有elements
Object remove(int i)	移除position "i"的element
Object get(int i)	傳回position "i"的element
Object set(int i, Object o)	將position "i"的element改為"o"
int indexOf(Object o)	傳回第一個"o"的position，若this list內沒有"o"，則傳回 -1
int lastIndexOf(Object o)	傳回最後一個"o"的position，若this list內沒有"o"，則傳回 -1
ListIterator listIterator()	傳回this的ListIterator object

Map

1. Map 沒有繼承 Collection，一個 Map 內的 elements 是 key-value pairs，其中 key 與 value 都必須是 reference type，且 key 必須是 unique
2. 特性是 non-orders (沒有順序性)，repetition allowed(不允許重複)

Object put(Object key, Object value)	將(key, value)放進this map，若指定的key已經存在，則新的value取代舊的value
Object get(Object key)	若this map含有參數"key"所指定的key-value pair，則傳回這個pair的value，否則傳回null
Object remove(Object key)	移除參數"key"所指定的mapping -- (key, value)
void clear()	移除所有的mapping -- (key, value)
int size()	傳回this map內，所含的key-value pairs個數
boolean isEmpty()	若this map是空的，則傳回true
boolean containsKey(Object key)	若this map含有參數"key"所指定的key-value pair，則傳回true
Set keySet()	傳回this map內，所有keys所組成的set，注意：所傳回的set是由this map所支持，對這個set所做的任何異動，同樣影響this map，反之亦是
Collection values()	傳回this map內，所有values所組成的collection。注意：所傳回的collection是由this map所支持，對這個collection所做的任何異動，同樣影響this map，反之亦是
Set entrySet()	傳回this map內，所有key-value pairs 所組成的set。注意：所傳回的set是由this map所支持，對這個set所做的任何異動，同樣影響this map，反之亦是

Java Collection 的 data structure

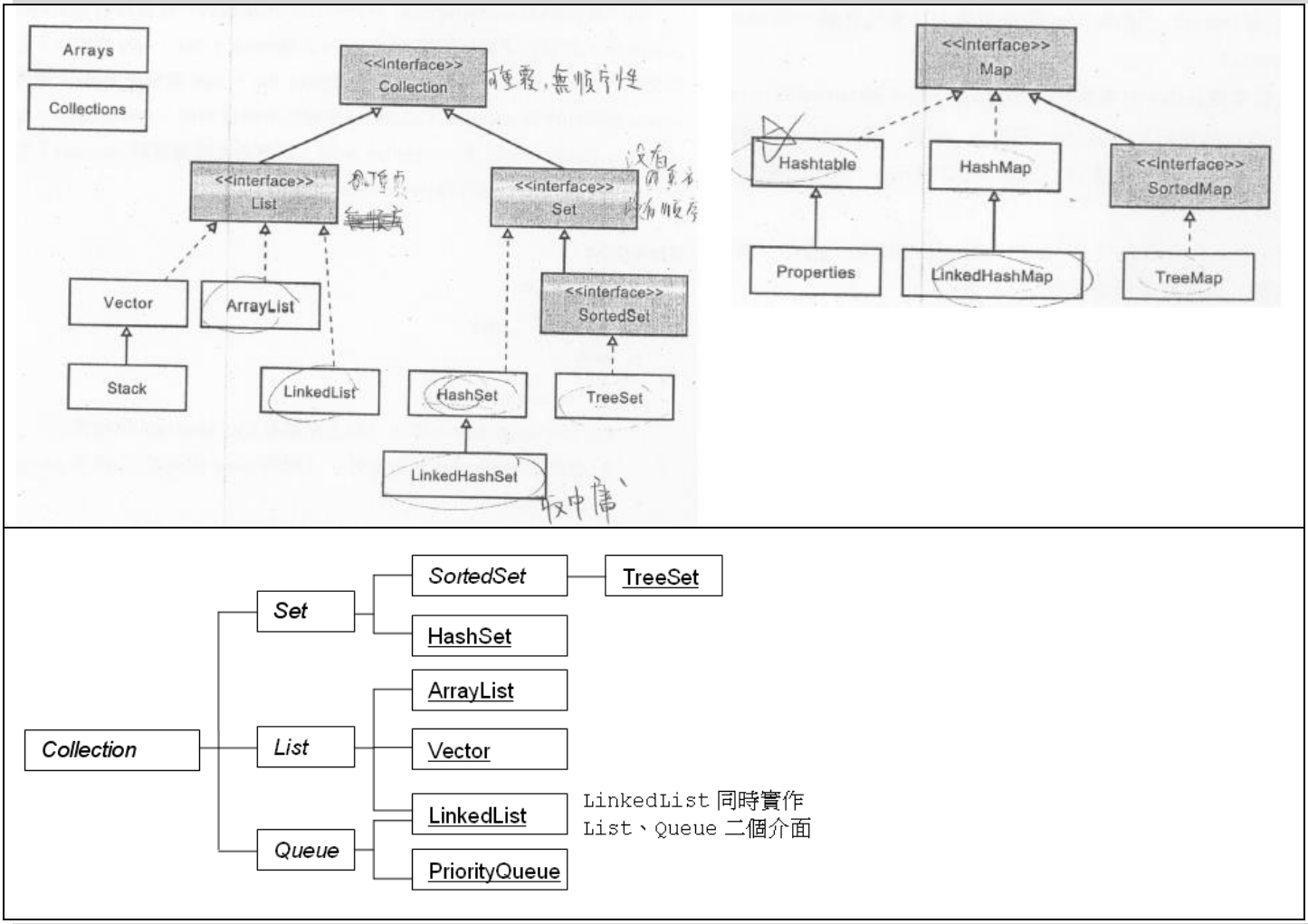
Array➔ array 的優點是 存取某個 element 的速度快。缺點是 resize 很不方便，insert 及 delete 某個 element 很不方便

Linked➔ Linked 的優點是很容易 resize、很容易就可 insert 或 remove 一個 element。缺點是存取某個 element 的速度慢

Tree➔Tree 優點是有 sort 的功能。缺點是存取的速度較慢

HashTable➔ HashTable 的優點存取資料速度快

Java Collection 的 Framework



ArrayList➔實作 List，資料結構 array，特性➔有順序性，允許重覆

1. 存取某個 element 的效率好。(因為 array 是使用 index 去 access element)，insert 或 remove 一個 element 的效率差。(須要移動一個 block 的 elements)
2. 不適合常需 resize 的環境 。(其實 array 是不能 reslze 的，resize 一個 array 的技巧是，當 array 需要 reslze 時，就 recreate 一個新的 array，並將舊 array 的資料 copy 到新的 array)
3. 不是 thread safe class 在 multi-threading 的環境下操作，須特別的機制，以保護資料的一致性。

LinkedList➔實作 List，資料結構 linked lis，特性➔有順序性，允許重覆

1. insert 或 remove 一個 element 的效率好 (只要更動兩個 elements 的 link)
2. 存取某個 elements 的效率差。(需要沿著 link，逐一的 search)
3. 不是 thread safe class

Queue 集合➔有順序性，允許重覆

1. Queue（佇列）集合元素存取的方式是先進先出（FIFO）
2. 一般建議使用專屬的 **offer()** 和 **poll()**方法，以避免產生不必要的例外事件，而造成元素存取中斷。
3. PriorityQueue 和 LinkedList 是其子類別：
   <1>. PriorityQueue：元素會先做升冪排序。
   <2>. LinkedList：元素加入的順序決定了存取的順序

Vector➔實作 List，資料結構 array，特性➔有順序性，允許重覆

1. 很像 ArrayList 但 Vector 的 critical methods 都是 synchronized。
2. 是 thread safe class 適合在 multithreading 的環境中使用，但執行速度較慢

Stack➔實作 List，資料結構 array，是 Vector 的子類別，特性➔有順序性，允許重覆

1. Stack（堆碟）集合元素存取的方式是先進後出（FILO）
2. 是 thread safe class
3. Stack 的重要 methods :

int search(Obiect o) //傳回 "o" 所在的 index，注意 index 是從上面算起，最上面的 index 是 1 不是 0

Obiect **push(Obiect item)**

Obiect **pop()** //傳回最上面的 elements 並 remove 它

Obiect **peek()** //傳回最上面的 elements 並不 remove 它

HashSet➔實作 Set，資料結構 hash table，特性➔沒有順序性，不允許重覆

1. 存取資料的效率好。(這是 hash table 的特性)
2. 不是 thread safe clåss 。

TreeSet➔實作 SortedSet，資料結構 balanced tree，特性➔遞增的順序，不允許重覆

1. 有 sorting 的功能
2. 但新增、移除資料的效率，比 HashSet 差。(這是 tree 的特性)
3. 不是 thread safe class

LinkedHashSet➔實作 Set，資料結構 hash table，double linked list，特性➔有順序性，不允許重覆

1. HashSet 的 subclass 0
2. Ordered 。
3. HashSet 的效率不錯，但它是 non-ordered，TreeSet 是 ordered ，但效率不好。LinkkedHashSet 取其中庸
4. 不是 thread safe class

HashMap➔實作 Map，資料結構 hash table，特性➔key 沒有順序性，key 不允許重覆

1. 存取資料的效率很好
2. 不是 thread safe class

TreeMap➔實作 SortedMap，資料結構 balanced tree，特性➔key 遞增的順序，key 不允許重覆

1. 有 sorting 的功能
2. 新增、移除資料的效率比 HashMap 差
3. 不是 thread safe class

LinkedHashMap➔實作 Map，資料結構 hash table 及 double linked list，特性➔key 有順序性，key 不允許重覆

1. HashMap 的 subclass 。
2. HashMap 的效率不錯，但它是 non-ordered，TreeMap 是 ordered，但效率不好。Linked HashMap 取其中庸
3. 不是 thread safe class。

HashTable➔實作 Map，資料結構 hash table，特性➔key 沒有順序性，key 不允許重覆

1. 很像 HashMap，但 HashTable 的 critical methods 都是 synchronized。
2. 是 thread safe class。適合在 mulithreading 的環境中使用，但執行的速度較慢。

Properties➔實作 Map，資料結構 hashtable，特性➔key 沒有順序性，key 不允許重覆

1. 是 HashTable 的 subclass
2. 是 thread safe çlass。適合在 mulithreading 的環境中使用，但執行的速度較慢。

```java
public static void List集合() {
    List list1 = new LinkedList();
    list1.add("aaa");
    list1.add("bbb");
    list1.add("ddd");
    list1.add("ccc");
    list1.add("bbb");
    list1.add("bbb");
    System.out.println("LinkedList==>" + list1);
    //-------------------------------------
    List<String> list2 = new ArrayList();
    list2.add("aaa");
    list2.add("bbb");
    list2.add("ddd");
    list2.add("ccc");
    list2.add("bbb");
    list2.add("bbb");
    System.out.println("ArrayList==>" + list2);
}
```

```
LinkedList==>[aaa, bbb, ddd, ccc, bbb, bbb]
ArrayList==>[aaa, bbb, ddd, ccc, bbb, bbb]
```

```java
public static void Stack集合() {
    //先進後出
    Stack<String> stack = new Stack();
    stack.push("1.orange");
    stack.push("2.apple");
    stack.push("3.banana");

    while (!stack.isEmpty()) {
        System.out.println(stack.pop() + " ");
    }
}
```

```
3.branana
2.apple
1.orange
```

```java
public static void Queue集合1() {

    Queue<String> pq = new PriorityQueue<String>();
    pq.offer("orange");
    pq.offer("apple");
    pq.offer("banana");

    System.out.println("toString()方式印==>" + pq);

    System.out.print("poll()方式印==>");
    while ((pq.peek()) != null) {
        System.out.print(pq.poll() + " ");
    }
    System.out.println();
}
```

```
toString()方式印==>[apple, orange, banana]
poll()方式印==>apple banana orange
```

```java
public static void Queue集合2() {

    Queue<String> pq = new LinkedList<String>();
    pq.offer("orange");
    pq.offer("apple");
    pq.offer("branana");

    System.out.println("toString()方式印==>" + pq);

    System.out.print("poll()方式印==>");
    while ((pq.peek()) != null) {
        System.out.print(pq.poll() + " ");
    }
    System.out.println();
}
```

```
toString()方式印==>[orange, apple, branana]
poll()方式印==>orange apple branana
```

```java
public static void Set集合1() {
    Set s1 = new HashSet();
    s1.add("aaa");
    s1.add("bbb");
    s1.add("ddd");
    s1.add("ccc");
    s1.add("bbb");
    s1.add("bbb");
    System.out.println("HashSet==>" + s1.toString());
    //-------------------------------------
    Set s2 = new TreeSet();
    s2.add("aaa");
    s2.add("bbb");
    s2.add("ddd");
    s2.add("ccc");
    s2.add("bbb");
    s2.add("bbb");
    System.out.println("TreeSet==>" + s2);
    //-------------------------------------
    Set s3 = new LinkedHashSet();
    s3.add("aaa");
    s3.add("bbb");
    s3.add("ddd");
    s3.add("ccc");
    s3.add("bbb");
    s3.add("bbb");
    System.out.println("LinkedHashSet==>" + s3);
}
```

```
HashSet==>[aaa, ccc, bbb, ddd]
TreeSet==>[aaa, bbb, ccc, ddd]
LinkedHashSet==>[aaa, bbb, ddd, ccc]
```

```java
public static void Set集合2() {
    List list1 = new LinkedList();
    list1.add("aaa");
    list1.add("bbb");
    list1.add("ddd");
    list1.add("ccc");
    list1.add("bbb");
    list1.add("bbb");
    System.out.println("LinkedList==>" + list1);
    //-------------------------------------
    Set set = new HashSet();
    set.addAll(list1);
    System.out.println("HashSet==>" + set);
}
```

```
LinkedList==>[aaa, bbb, ddd, ccc, bbb, bbb]
HashSet==>[aaa, ccc, bbb, ddd]
```

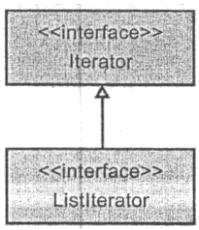
```java
public static void Map集合() {
    Map m1 = new HashMap();
    m1.put("aaa", "a1");
    m1.put("bbb", "a2");
    m1.put("ddd", "a4");
    m1.put("ccc", "a3");
    m1.put("aaa", "a5");
    m1.put("bbb", "a6");
    System.out.println("HashMap==>" + m1);
    //-------------------------------------
    Map m2 = new TreeMap();
    m2.put("aaa", "a1");
    m2.put("bbb", "a2");
    m2.put("ddd", "a4");
    m2.put("ccc", "a3");
    m2.put("aaa", "a5");
    m2.put("bbb", "a6");
    System.out.println("TreeMap=>" + m2);
    //-------------------------------------
    Map m3 = new LinkedHashMap();
    m3.put("aaa", "a1");
    m3.put("bbb", "a2");
    m3.put("ddd", "a4");
    m3.put("ccc", "a3");
    m3.put("aaa", "a5");
    m3.put("bbb", "a6");
    System.out.println("LinkedHashMap==>" + m3);

    Map m4 = new HashMap();
    m4.putIfAbsent("aaa", "a1");
    m4.putIfAbsent("bbb", "a2");
    m4.putIfAbsent("ddd", "a4");
    m4.putIfAbsent("ccc", "a3");
    m4.putIfAbsent("aaa", "a5"); //防止 key值 覆盖
    m4.putIfAbsent("bbb", "a6");
    System.out.println("HashMap==>" + m4);
}
```

```
HashMap==>{aaa=a5, ccc=a3, bbb=a6, ddd=a4}
TreeMap=>{aaa=a5, bbb=a6, ccc=a3, ddd=a4}
LinkedHashMap==>{aaa=a5, bbb=a6, ddd=a4, ccc=a3}
HashMap==>{aaa=a1, ccc=a3, bbb=a2, ddd=a4}
```

Iterator(佚代器)

1. Iterator 是一個 interface，任何直接或間接 implements Collection interface 的 collection (如 ArrayList 、LinkList、Hash Set...)，都會有一 個 method ➔ iterator()
2. 這個 method 會傳回 this collection 的 Iterator object ，這個 Iterator object 內含 this collection 的所有 elements 而且這個 Iterator object 保證已實作了 Iterator interface
3. 我們可以用 Iterator interface 所宣告的 methods 來逐一列舉出所有的 elements，所列舉出 elements 的順序，決定於 this collection 的特性



boolean hasNext()	若iterator內還有elements尚未被列舉，則return true。
Object next()	傳回下一個element。注意：return type是Object，大多時候，這個所傳回的object，必須downward cast到它真正的type。
void remove()	把剛才next() method所傳回的element從collection中移除。 注意一：不是從iterator object中移除，而是從這iterator的collection中移除。 注意二：每次呼叫remove() method前，一定得先呼叫next() method。

Map 如何使用 Iterator ➔列舉 Map 的 elements 有三種方法

1. 任何的 set，list 可以用統一的介面 iterator( )，來列舉出它所有的 elements
2. map 內所有 keys 所成的集合是一個 Set，因為 key 是 unique，使用 keyset( )可以取出 this map 的 key set
3. map 內所有 values 所成的集合是一個 collection，因為 map 內的 value 可以重復，使用 values( )這個 method( )，可以取出 this map 內的 value collection
4. 一個 map 內所有 key-value pairs 所成的集合是一個 set 因為 key 的 mapping 是唯一的(一個 key 最多只能對應到一個 value) 使用 entryset( )這個 method( )可以取出 this map 內的 entry set
5. entry set 內的 element 是 key-value pair 的，變成 **Iterator 物件**後，Java 特別為這個 **Iterator 物件**的 key-value pair 定義一個 interface➔Map.Entry ，Map.Entry 是 Map interface 的 inner interface( ) ，Map.Entry 所宣告的重要 methods 如下：

Object getKey()	傳回這個entry的key
Object getValue()	傳回這個entry的value
Object setValue(Object v)	用"v"取代這個entry的value

```java
public static void iterator1() {
    Set set = new HashSet();
    set.add("aaa");
    set.add("bbb");
    set.add("ddd");
    set.add("ccc");
    set.add("bbb");
    set.add("bbb");
    System.out.println(set);

    Iterator it1 = set.iterator();
    System.out.println(it1);

    while (it1.hasNext()) {
        Object o = it1.next();
        System.out.print(o + " ");
        it1.remove();
    }
    System.out.println();
}
```

```
[aaa, ccc, bbb, ddd]
java.util.HashMap$KeyIterator@15db9742
aaa ccc bbb ddd
```

```java
public static void iterator2() {
    List list = new LinkedList();
    list.add("aaa");
    list.add("bbb");
    list.add("ddd");
    list.add("ccc");
    list.add("bbb");
    list.add("bbb");
    System.out.println(list);

    Iterator it1 = list.iterator();
    while (it1.hasNext()) {
        Object o = it1.next();
        System.out.print(o + " ");
    }
    System.out.println();
}
```

```
[aaa, bbb, ddd, ccc, bbb, bbb]
aaa bbb ddd ccc bbb bbb
```

```java
public static void iterator3() {
    Map m = new HashMap();
    m.put("aaa", "a1");
    m.put("bbb", "a2");
    m.put("ddd", "a4");
    m.put("ccc", "a3");
    m.put("aaa", "a5");
    m.put("bbb", "a6");
    System.out.println(m);

    System.out.println("key------------------------");
    Set key = m.keySet();
    Iterator it1 = key.iterator();
    while (it1.hasNext()) {
        Object o = it1.next();
        System.out.print(o + " ");
    }
    System.out.println();
    System.out.println("value------------------------");
    Collection value = m.values();
    Iterator it2 = value.iterator();
    while (it2.hasNext()) {
        Object o = it2.next();
        System.out.print(o + " ");
    }
    System.out.println();
    System.out.println("entryset------------------------");
    Set entryset = m.entrySet();
    Iterator it3 = entryset.iterator();
    while (it3.hasNext()) {
        Map.Entry o = (Map.Entry) it3.next();
        Object ok = o.getKey();
        Object ov = o.getValue();
        System.out.println(ok + " " + ov);
    }
}
```

```
{aaa=a5, ccc=a3, bbb=a6, ddd=a4}
key------------------------
aaa ccc bbb ddd
value------------------------
a5 a3 a6 a4
entryset------------------------
aaa a5
ccc a3
bbb a6
ddd a4
```

```java
public static void 印集合的方式() {
    List list = new LinkedList();
    list.add("aaa");
    list.add("bbb");
    list.add("ccc");
    list.add("ddd");
    list.add("ccc");
    list.add("ddd");
    //1.直接印
    System.out.println(list.toString());
    System.out.println();
    //2.快捷迴圈
    for (Object o : list) {
        System.out.print(o + " ");
    }
    System.out.println();
    //3.產生到陣列印
    for (Object o : list.toArray()) {
        System.out.print(o + " ");
    }
    System.out.println();
    //4.產生到ierator的型態
    Iterator it = list.iterator();
    while (it.hasNext()) {
        Object o = it.next();
        System.out.print(o + " ");
    }
    //5.直接印 Iterator ==>會得到 null
    System.out.println();
    System.out.println(it);
    //6.Iterator 型態不能用快捷迴圈印
    for (Object o : it) {
        System.out.print(o);
    }
}
```

```
[aaa, bbb, ccc, ddd, ccc, ddd]

aaa bbb ccc ddd ccc ddd
aaa bbb ccc ddd ccc ddd
aaa bbb ccc ddd ccc ddd
java.util.LinkedList$ListItr@15db9742
```

Arrays 與 Collections

Arrays 是一個 class 它定義一些有用的 static methods 用來處理 array·Collections 也是一個 class (不要與 interface Collection 搞混) ，它亦定義 一些有用的 static methods 用來處理 co11ection

```
public static void Arrays_排序() {
    //排序
    String[] sa = {"a", "abc", "xyz", "ijk", "ab"};
    Arrays.sort(sa);
    for (String i : sa) {
        System.out.print(i + " ");
    } // a ab abc ijk xyz
    System.out.println();

    //布林不能排序 會 Compiler 錯
    boolean[] ar3 = {true, false};
    //Arrays.sort(ar3);
}
```

```
a ab abc ijk xyz
```

```
public static void Arrays_陣列放到集合() {
    String[] sa = {"a", "abc", "xyz", "ijk", "ab"};

    //把陣列放到集合
    List list1 = Arrays.asList(sa); //固定size
    List list2 = new ArrayList(Arrays.asList(sa)); //非固定size

    // list1.add("yyy"); //固定 size不能再加入 , 會當掉
    System.out.print(list1);
    System.out.println();

    list2.add("yyy"); // 非固定 size 可再加入
    System.out.print(list2);
    System.out.println();
}
```

```
[a, abc, xyz, ijk, ab]
[a, abc, xyz, ijk, ab, yyy]
```

```
public static void Arrays_比較相不相等() {
    String[] sa1 = {"A", "B", "C", "D"};
    String[] sa2 = {"A", "B", "C", "D"};

    //陣列比較 String 有覆寫 equals 但 String[] 沒有
    System.out.println("sa1 == sa2 是" + (sa1 == sa2));  //false
    System.out.println("sa1.equals(sa2) 是 " + (sa1.equals(sa2)));  //false
    System.out.println("Arrays.equals(sa1, sa2) 是" + (Arrays.equals(sa1, sa2)));  //true
}
```

```
sa1 == sa2 是false
sa1.equals(sa2) 是 false
Arrays.equals(sa1, sa2) 是true
```

```
public static void Arrays_binarysearch1() {
    int index;
    String[] ar1 = {"3", "1", "5", "4", "7"};

    Arrays.sort(ar1);
    for (String i : ar1) {
        System.out.print(i + " ");
    }
    System.out.println();

    //1 3 4 5 7
    index = Arrays.binarySearch(ar1, "5");
    System.out.println("index=" + index); //3
    index = Arrays.binarySearch(ar1, "6");
    System.out.println("index=" + index);  //-5
}
```

```
1 3 4 5 7
index=3
index=-5
```

```
public static void Arrays_binarysearch2() {
    int index;
    String[] sa = {"blue", "red", "green", "yellow", "orange"};

    Arrays.sort(sa);
    for (String i : sa) {
        System.out.print(i + " ");
    }
    System.out.println();

    //blue green orange red yellow

    index = Arrays.binarySearch(sa, "orange");
    System.out.println("index=" + index);  //2
    index = Arrays.binarySearch(sa, "violet");
    System.out.println("index=" + index);  //-5
}
```

```
blue green orange red yellow
index=2
index=-5
```

```java
public static void Collections_排序() {
    List list = new LinkedList();
    list.add("a");
    list.add("c");
    list.add("b");
    list.add("d");
    list.add("c");
    Collections.sort(list);
    System.out.println(list);
}
```

```
[a, b, c, c, d]
```

```java
public static void Collections_洗牌() {
    List list = new LinkedList();
    list.add("a");
    list.add("c");
    list.add("b");
    list.add("d");
    list.add("c");
    //洗牌
    Collections.shuffle(list);
    System.out.println(list);
}
```

```
[c, c, b, a, d]
```

排序物件

Sorting 一定會比較兩個 elements 的大小。除了 boolean 之外，另外 7 種 primitive types 都可以比較大小，但 objects 如何來比較大小?有兩種方法：

1. natural order (自然順序)
   <1>. 兩個不同 objects 的 natural order 交由 Comparable 介面 定義依 Java API documents 的規定
   它的傳回值 若 this 比 o 大 傳回正值，若 this 比 o 小 傳回負值，若 this==o 傳回 0
   <2>. Comparable 內宣告了 一個方法
   int compareto(Object o)
   <3>. String 及 7 個 primitive types 的 wrapper class (Boolean 除外) 都 有 implements Comparable interface
2. arbitrary order (武斷順序)
   <1>. 兩個不同 objects 的 arbitrary order 交由 Comparator 介面 定義，誰大誰小交給裁判 comparator 判定
   它的傳回值 若 o1 比 o2 大 傳回正值，若 o1 比 o2 小 傳回負值，若 o1==o2 傳回 0
   <2>. Comparator 內宣告了 兩個方法
   boolean equals(Object o)➔一個實作 Comparator interface 的 class 並不需實作 equals 因為 Object 已實作它了
   int compare(Object o1，Object o2)

Arrays 內的兩個常用的方法

Arrays.sort(Object[] oa)➔根據自然順序➔所以 array 內 的所有 elements 必須 implements Comparable interface

Arrays.sort(Object[] oa，Comparator o)➔根據武斷順序 comparator --"c" 必須 implements Comparator interface

```java
//七個包裝類別 與 字串 已有複寫 Comparable 所以可以排序
    public static void 字串可以排序_有實作Comparable() {
        Comparator p = new Comparator<String>() {
            public int compare(String m, String n) {
                return n.compareTo(m);
            }
        };
        String[] sa = {"b", "c", "a"};

        Arrays.sort(sa);
        for (String a : sa) {
            System.out.print(a + " ");
        }
        System.out.println();
        Arrays.sort(sa, p);
        for (String a : sa) {
            System.out.print(a + " ");
        }
        System.out.println();
    }
```

```
a b c
c b a
```

```java
public static void 七個包裝類別可以排序_有實作Comparable() {
    Comparator p = new Comparator<Integer>() {
        public int compare(Integer m, Integer n) {
            return n.compareTo(m);
        }
    };
    Integer[] sa = {4, 2, 3, 1};

    Arrays.sort(sa);
    for (Integer a : sa) {
        System.out.print(a + " ");
    }
    System.out.println();
    Arrays.sort(sa, p);
    for (Integer a : sa) {
        System.out.print(a + " ");
    }
    System.out.println();
}
```

```
1 2 3 4
4 3 2 1
```

```java
//自定的類別無法排序 , 除非實作 自然順序 compatable的 compareTo  //會當掉
public static void 自訂的類別無法排序_沒有實作Comparable() {

    CollectionDemo1[] sa = {new CollectionDemo1(4), new CollectionDemo1(2), new CollectionDemo1(3)
            , new CollectionDemo1(1)};

    Arrays.sort(sa);
    for (CollectionDemo1 a : sa) {
        System.out.print(a + " ");
    }
}
```

```
java.lang.ClassCastException
```

```java
public static void 自訂的類別_可以排序有實作Comparable與Comparator_陣列_沒有泛型() {
    CollectionDemo2 p = new CollectionDemo2();
    CollectionDemo2[] sa = {new CollectionDemo2(4), new CollectionDemo2(2),
        new CollectionDemo2(3), new CollectionDemo2(1)};
    Arrays.sort(sa, null);
    for (CollectionDemo2 x : sa) {
        System.out.print(x + " ");
    }
    System.out.println();
    Arrays.sort(sa, p);
    for (CollectionDemo2 x : sa) {
        System.out.print(x + " ");
    }
    System.out.println();
}
```

```
1 2 3 4
4 3 2 1
```

```java
public static void 自訂的類別_可以排序有實作Comparable與Comparator_集合_沒有泛型() {
    CollectionDemo2 p = new CollectionDemo2();
    List list = new ArrayList();
    list.add(new CollectionDemo2(4));
    list.add(new CollectionDemo2(2));
    list.add(new CollectionDemo2(3));
    list.add(new CollectionDemo2(1));

    Collections.sort(list, null);
    for (Object obj : list) {
        System.out.print(obj + " ");
    }
    System.out.println();
    Collections.sort(list, p);
    for (Object obj : list) {
        System.out.print(obj + " ");
    }
    System.out.println();
}
```

```
1 2 3 4
4 3 2 1
```

```java
public static void 自訂的類別_可以排序有實作Comparator_陣列_有泛型() {
    CollectionDemo3 p = new CollectionDemo3();
    CollectionDemo3[] sa = {new CollectionDemo3(4), new CollectionDemo3(2),
        new CollectionDemo3(3), new CollectionDemo3(1)};
    Arrays.sort(sa, null);
    for (CollectionDemo3 x : sa) {
        System.out.print(x + " ");
    }
    System.out.println();

    Arrays.sort(sa, p);

    for (CollectionDemo3 x : sa) {
        System.out.print(x + " ");
    }
    System.out.println();
}
```

```java
public static void 自訂的類別_可以排序有實作Comparator_集合_有泛型() {
    CollectionDemo3 p = new CollectionDemo3();
    List<CollectionDemo3> list = new ArrayList();
    list.add(new CollectionDemo3(4));
    list.add(new CollectionDemo3(2));
    list.add(new CollectionDemo3(3));
    list.add(new CollectionDemo3(1));

    Collections.sort(list, null);

    for (CollectionDemo3 obj : list) {
        System.out.print(obj + " ");
    }
    System.out.println();

    Collections.sort(list, p);
    for (CollectionDemo3 obj : list) {
        System.out.print(obj + " ");
    }
    System.out.println();
}
```

```java
public static void 不同資料型態不能sort() {
    //布林不能排序 , 會 Compiler 錯誤
     boolean[] ar3 = {true, false};
     Arrays.sort(ar3);
    //不同資料型態不能sort , 會當掉
    Object[] myObjects = {
        new Integer(12),
        new String("foo"),
        new Integer(5), // new Boolean(true)
    };
    Arrays.sort(myObjects);
}
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
        at java.lang.String.compareTo(String.java:108)
        at java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort.java:290)
2 4     at java.util.ComparableTimSort.sort(ComparableTimSort.java:157)
        at java.util.ComparableTimSort.sort(ComparableTimSort.java:146)
        at java.util.Arrays.sort(Arrays.java:472)
        at java16_集合.Test16.不同資料型態不能sort(Test16.java:395)
        at java16_集合.Main.main(Main.java:26)
Java Result: 1
```

hashCode( ) 方法➔禁止重複

1. Java Collection API 內有許多 collection 所使用的儲存技術是 Hash table 如 HashTable 、HashSet、 HashMap 、LinkedHashSet 及 LinkedHashMap
2. 這種 storage technique 是根據 object 的 hashCode values 來決定它的儲存位置。而一個 object 的 hashCode value 是由它的 hashCode() method 來決定 ，Object 已經定義了 hashCode( )的方法 ➔ public int hashCode()
3. 這個方法的 return value 必須確保一件事 equals object must have equals hash codes
   即若兩個 object 經由 equals(Object obj) 的判定為 “相等” ，則這兩個 object 的 hashCode( )方法必須傳回相同的整數值，否則這種 object 在以 hash table 為儲存技術的 collection 上操作，會發生錯誤。
   所以 若 overrides equals(Object o) methd 也必須 overrides hashCode( ) method
   若 o1.equals(o2)= =true 則 o1.hashCode()= =o2.hashCode() 也必須為 true
4. String 與 八 個 primitive types 的 wrapper classes 都已 overrides equals( ) method，也都正確的 overrides hashCode() method
5. 一個簡單，但合乎要求的 hashCode( ) 實作是將所有重要 elements 的 hashCode values 做 bitwise XOR

```
public static void 八個包裝類別_hashCode的值() {

    Integer a = new Integer(4);
    Integer b = new Integer(4);

    System.out.println(a.hashCode());
    System.out.println(b.hashCode());
}
```

```
4
4
```

```
public static void 字串_hashCode的值() {

     String a = "abc";
     String b = "abc";

     System.out.println(a.hashCode());
     System.out.println(b.hashCode());
 }
```

```
96354
96354
```

```
public static void 自訂類別_沒有覆寫hashCode的值() {

    CollectionDemo2 a = new CollectionDemo2(4);
    CollectionDemo2 b = new CollectionDemo2(4);

    System.out.println(a.hashCode());
    System.out.println(b.hashCode());
 }
```

```
20785379
4677928
```

```
public static void 自訂類別_有覆寫hashCode的值() {

    CollectionDemo4 a = new CollectionDemo4(4);
    CollectionDemo4 b = new CollectionDemo4(4);
    System.out.println(a.hashCode());
    System.out.println(b.hashCode());
 }
```

```
4
4
```

```
//八個包裝類別與字串已有複寫 equals 與 hashcode()

public static void 八個包裝類別_加到HashSet集合() {
    Integer a = new Integer(4);
    Integer b = new Integer(4);

    System.out.println(a.hashCode());  //呼叫到 Object 若沒覆寫
    System.out.println(b.hashCode());

    Set aSet = new HashSet();
    aSet.add(a);
    aSet.add(b);
    System.out.println(aSet);
}
```

```
4
4
[4]
```

```java
public static void 字串_加到HashSet集合() {
    String a = "123";
    String b = "123";

    System.out.println(a.hashCode());  //呼叫到 Object 若沒覆寫
    System.out.println(b.hashCode());

    Set aSet = new HashSet();
    aSet.add(a);
    aSet.add(b);
    System.out.println(aSet);
}
```

```
48690
48690
[123]
```

```java
public static void 自訂類別_沒有覆寫equals與hashCode_加到HashSet集合() {
    CollectionDemo2 a = new CollectionDemo2(4);
    CollectionDemo2 b = new CollectionDemo2(4);
    System.out.println(a.hashCode());  //呼叫到 Object 若沒覆寫
    System.out.println(b.hashCode());

    Set aSet = new HashSet();
    aSet.add(a);
    aSet.add(b);
    System.out.println(aSet);
}
```

```
20785379
4677928
[4, 4]
```

```java
public static void 自訂類別_有覆寫equals與hashCode_加到HashSet集合() {
    CollectionDemo4 a = new CollectionDemo4(4);
    CollectionDemo4 b = new CollectionDemo4(4);
    System.out.println(a.hashCode());
    System.out.println(b.hashCode());

    Set aSet = new HashSet();
    aSet.add(a);
    aSet.add(b);
    System.out.println(aSet);
}
```

```
4
4
[4]
```

```java
public static void 覆寫hashCode的演算法() {

    CollectionDemo5 a = new CollectionDemo5(1, 1);
    CollectionDemo5 b = new CollectionDemo5(1, 2);
    System.out.println(a.hashCode());
    System.out.println(b.hashCode());

    Set aSet = new HashSet();
    aSet.add(a);
    aSet.add(b);
    System.out.println(aSet);
}
```

```
2
3
[1, 1]
```

```
//七個包裝類別跟字串已有實作 Comparable ,所以可以禁止重覆與排序
//傳回 0 就代表相等
    public static void 七個包裝類別_加到TreeSet集合() {

        Integer a = new Integer(4);
        Integer b = new Integer(6);
        Integer c = new Integer(4);

        Set aSet = new TreeSet();
        aSet.add(a);
        aSet.add(b);
        aSet.add(c);
        System.out.println(aSet);
    }
```

```
[4, 6]
```

```
public static void 字串_加到TreeSet集合() {

    String a = "123";
    String b = "123";
    String c = "abc";

    Set aSet = new TreeSet();
    aSet.add(a);
    aSet.add(b);
    aSet.add(c);
    System.out.println(aSet);
}
```

```
[123, abc]
```

```
public static void 自訂類別_沒有實作Comparable_加到TreeSet集合() {
    CollectionDemo2 a = new CollectionDemo2(4);
    CollectionDemo2 b = new CollectionDemo2(6);
    CollectionDemo2 c = new CollectionDemo2(4);

    Set aSet = new TreeSet();
    aSet.add(a);
    aSet.add(b);
    aSet.add(c);
    System.out.println(aSet);
}
```

```
java.lang.ClassCastException:
```

```
public static void 自訂類別_有實作Comparable_加到TreeSet集合() {

    CollectionDemo3 a = new CollectionDemo3(4);
    CollectionDemo3 b = new CollectionDemo3(6);
    CollectionDemo3 c = new CollectionDemo3(4);
    Set aSet = new TreeSet();
    aSet.add(a);
    aSet.add(b);
    aSet.add(c);
    System.out.println(aSet);
}
```

```
[4, 6]
```

```java
public static void 自訂類別_沒有覆寫hashCode_加到Map() {
    Map m = new HashMap();
    m.put(new CollectionDemo6(170, 66), "peter");
    System.out.println(m.get(new CollectionDemo6(170, 66)));
}
```

```
null
```

```java
public static void 自訂類別_有覆寫hashCode_加到Map() {
    Map m = new HashMap();
    m.put(new CollectionDemo7(170, 66), "peter");
    System.out.println(m.get(new CollectionDemo7(170, 66)));
}
```

```
peter
```

```java
public static void 自訂類別_有實作comparable() {
    Map m = new TreeMap();
    m.put(new CollectionDemo8(170, 76), "peter");
    m.put(new CollectionDemo8(150, 56), "carl");
    m.put(new CollectionDemo8(170, 76), "albert");
    System.out.print(m);
}
```

```
{150 56=carl, 170 76=albert}
```

```java
public static void 自訂類別_有覆寫equals跟hashCode() {
    Map m = new HashMap();
    m.put(new CollectionDemo8(170, 76), "peter");
    m.put(new CollectionDemo8(150, 56), "carl");
    m.put(new CollectionDemo8(170, 76), "albert");
    System.out.print(m);
}
```

```
{170 76=albert, 150 56=carl}
```

```java
public class CollectionDemo1 {

    public int x;

    public CollectionDemo1(int x) {
        this.x = x;
    }

    public String toString() {
        return String.valueOf(x);
    }
}
```

```java
class CollectionDemo2 implements Comparable, Comparator {

    public int x;

    public CollectionDemo2() {
    }

    public CollectionDemo2(int x) {
        this.x = x;
    }

    public String toString() {
        return Integer.toString(x);
    }

    public int compareTo(Object obj) { //由小到大排
        CollectionDemo2 a = (CollectionDemo2) obj;
        if (x < a.x) {
            return -1;
        }
        if (x > a.x) {
            return 1;
        }
        return 0;
    }

    public int compare(Object obj1, Object obj2) { //由大到小排-裁判

        CollectionDemo2 a = (CollectionDemo2) obj1;
        CollectionDemo2 b = (CollectionDemo2) obj2;
        if (a.x < b.x) {
            return 1;
        }
        if (a.x > b.x) {
            return -1;
        }
        return 0;
    }
}
```

```java
class CollectionDemo3 implements Comparable<CollectionDemo3>, Comparator<CollectionDemo3> {

    public int x;

    public CollectionDemo3() {
    }

    public CollectionDemo3(int x) {
        this.x = x;
    }

    public String toString() {
        return Integer.toString(x);
    }

    public int compareTo(CollectionDemo3 obj) { //由小到大排
        CollectionDemo3 a = obj;
        if (x < a.x) {
            return -1;
        }
        if (x > a.x) {
            return 1;
        }
        return 0;
    }

    public int compare(CollectionDemo3 obj1, CollectionDemo3 obj2) { //由大到小排-裁判

        CollectionDemo3 a = obj1;
        CollectionDemo3 b = obj2;
        if (a.x < b.x) {
            return 1;
        }
        if (a.x > b.x) {
            return -1;
        }
        return 0;
    }
}
```

```java
class CollectionDemo4 {

    public int x;

    public CollectionDemo4(int x) {
        this.x = x;
    }
    public String toString() {
        return Integer.toString(x);
    }
    public boolean equals(Object obj) {
        if ((obj != null && obj instanceof CollectionDemo4)) {
            if ((x == ((CollectionDemo4) obj).x)) {
                return true;
            }
        }
        return false;
    }
    public int hashCode() {
        return x;
    }
}
```

```java
class CollectionDemo5 {

    private int x;
    private int y;

    public CollectionDemo5(int x, int y) {
        this.x = x;
        this.y = y;
    }

// equals 只拿 x 來比較，所以 hashcode 也只能拿 x 來產生值
    public boolean equals(Object obj) {
        if ((obj != null) && (obj instanceof CollectionDemo5)) {
            if (x == ((CollectionDemo5) obj).x) {
                return true;
            }
        }
        return false;
    }

    public String toString() {
        return Integer.toString(x);
    }
//    public int hashCode() { //ok
//        return x;
//    }
//    public int hashCode() { //不行
//        return y;
//    }
//
//    public int hashCode() { // ok
//        return 1;
//    }
//

    public int hashCode() { //不行
        return x + y;
    }
}
```

```java
class CollectionDemo6 {

    private int height;
    private int weight;

    public CollectionDemo6(int height, int weight) {
        this.height = height;
        this.weight = weight;
    }

    public String toString() {
        return height + " " + weight;
    }

    public boolean equals(Object obj) {
        if ((obj != null && obj instanceof CollectionDemo6)) {
            if ((height == ((CollectionDemo6) obj).height)
                    && (weight == ((CollectionDemo6) obj).weight)) {
                return true;
            }
        }
        return false;
    }
    // object hashcode()
}
```

```java
class CollectionDemo7 {

    private int height;
    private int weight;

    public CollectionDemo7(int height, int weight) {
        this.height = height;
        this.weight = weight;
    }
    public String toString() {
        return height + " " + weight;
    }
    public boolean equals(Object obj) {
        if ((obj != null && obj instanceof CollectionDemo7)) {
            if ((height == ((CollectionDemo7) obj).height)
                    && (weight == ((CollectionDemo7) obj).weight)) {
                return true;
            }
        }
        return false;
    }
    public int hashCode() {
        return (new Integer(height).hashCode())
                ^ (new Integer(weight).hashCode());
    }
}
```

```
class CollectionDemo8 implements Comparable {

    private int height;
    private int weight;

    public CollectionDemo8(int height, int weight) {
        this.height = height;
        this.weight = weight;
    }
    public String toString() {
        return height + " " + weight;
    }
    public boolean equals(Object obj) {
        if ((obj != null) && (obj instanceof CollectionDemo8)) {
            if ((height == ((CollectionDemo8) obj).height)
                    && (weight == ((CollectionDemo8) obj).weight)) {
                return true;
            }
        }
        return false;
    }
    public int hashCode() {
        return (new Integer(height).hashCode())
                ^ (new Integer(weight).hashCode());
    }
    public int compareTo(Object obj) {
        CollectionDemo8 a = (CollectionDemo8) obj;
        if (height < a.height) {
            return -1;
        }
        if (height > a.height) {
            return 1;
        }
        if (weight < a.weight) {
            return -1;
        }
        if (weight > a.weight) {
            return 1;
        }
        return 0;
    }
}
```

```
class CollectionDemo9 {

    private String name;
    private int age;
    private double weight;

    public CollectionDemo9(String name, int age, int weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }

    public boolean equals(Object obj) {
        if ((obj != null) && (obj instanceof CollectionDemo9)) {
            CollectionDemo9 t = (CollectionDemo9) obj;

            if ((name.equals(t.name))
                    && (age == t.age)
                    && (weight == t.weight)) {
                return true;
            }
        }
        return false;
    }

    public int hashCode() {
        int i = (name.hashCode())
                ^ (new Integer(age).hashCode())
                ^ (new Double(weight).hashCode());
        return i;
    }
}
```