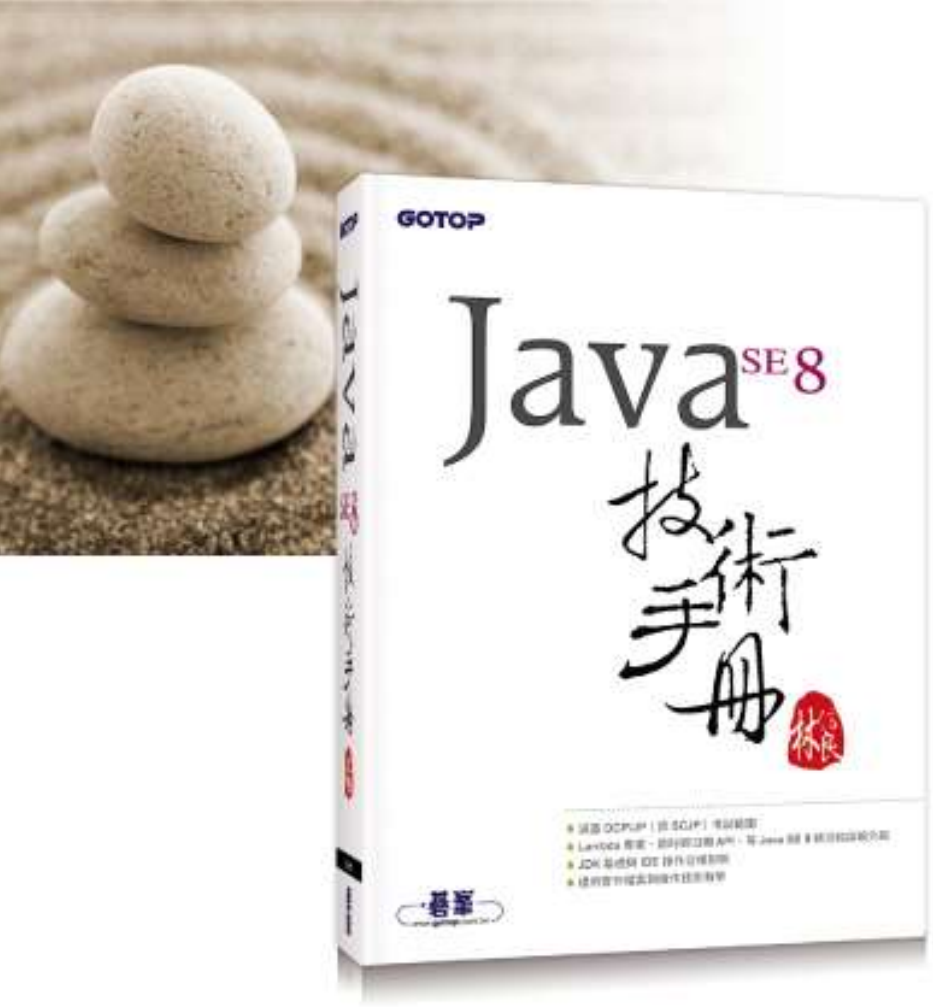


Java SE 8 技術手冊

林信良

- 涵蓋 OCP/JP（原 SCJP）考試範圍
- Lambda 專案、新時間日期 API、等 Java SE 8 新功能詳細介紹
- JDK 基礎與 IDE 操作交相對照
- 提供實作檔案與操作錄影教學

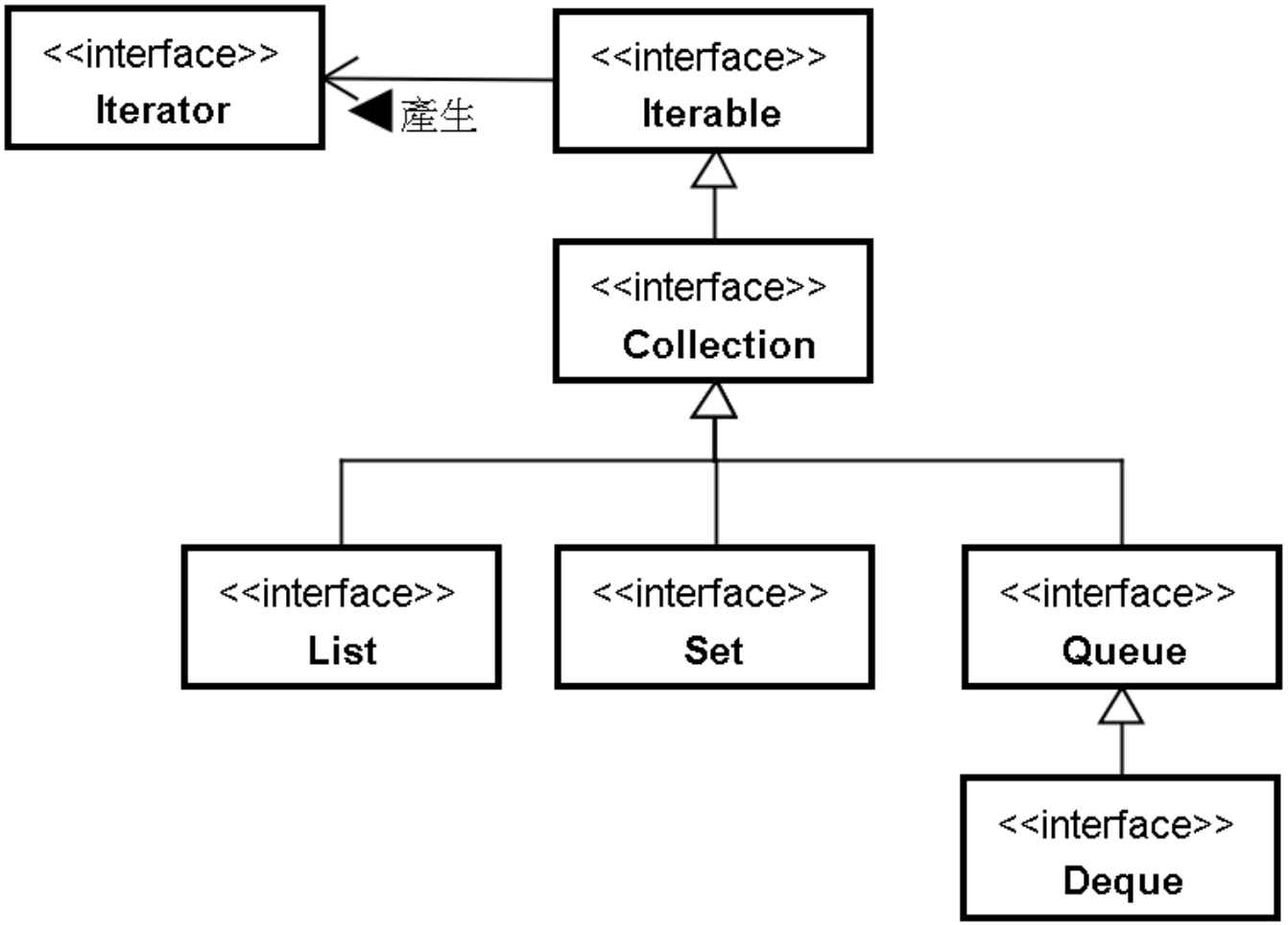
碁峰資訊

Collection與Map

學習目標

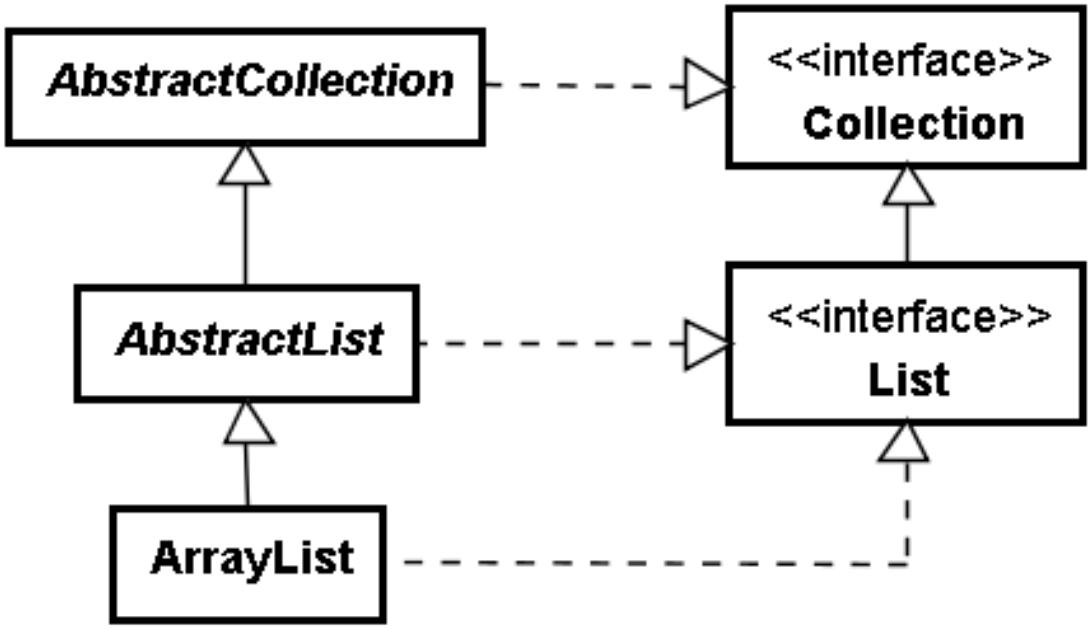
- 認識Collection與Map架構
- 使用Collection與Map實作物件
- 對收集之物件進行排序
- 簡介Lambda表示式
- 簡介泛型語法

認識Collection架構



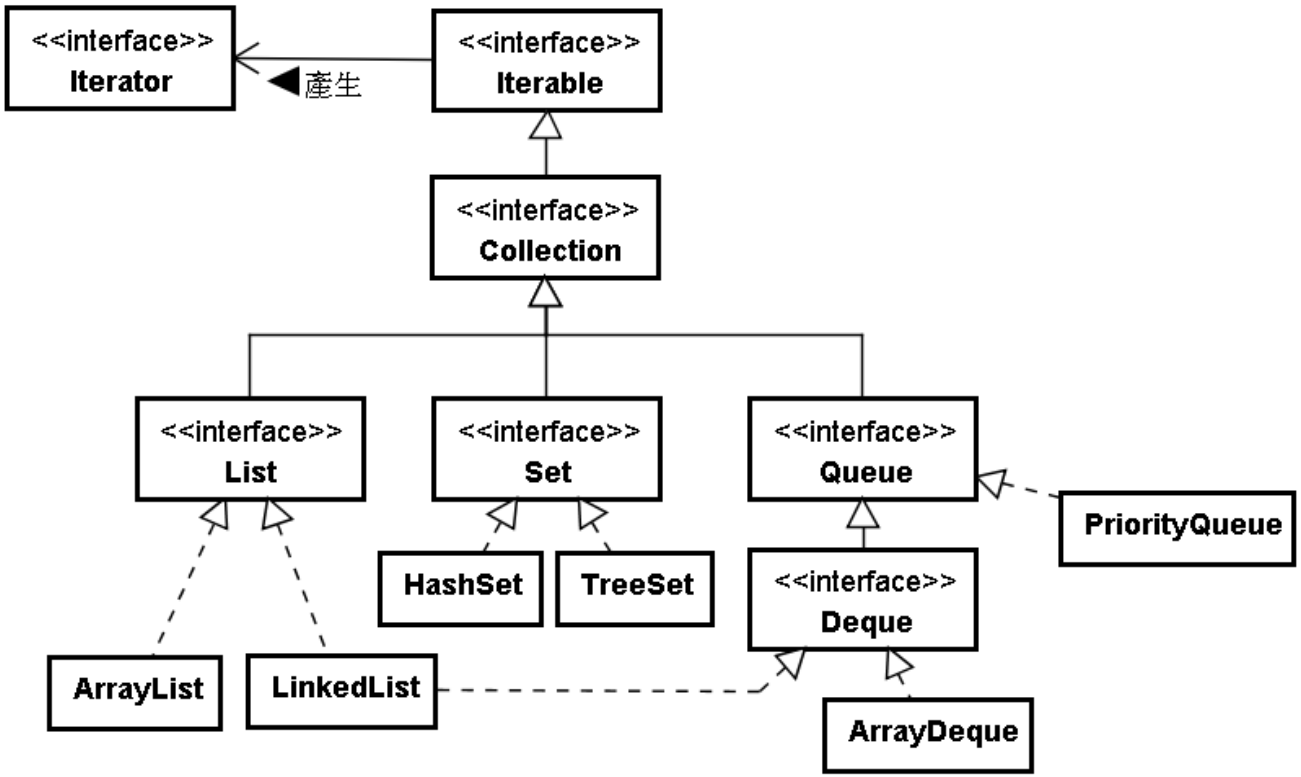
認識Collection架構

- 如果想要收集時具有索引順序，實作方式之一就是使用陣列，而以陣列實作List的就是java.util.ArrayList



認識Collection架構

- 有時為了只表示我們感興趣的介面或類別，會簡化繼承與實作架構圖



具有索引的List

- List是一種Collection，作用是收集物件，並以索引方式保留收集的物件順序
- 實作類別之一是**java.util.ArrayList**，其實作原理大致如6.2.5的ArrayList範例

```
public static void main(String[] args) {
    List names = new ArrayList();
    collectNameTo(names);
    out.println("訪客名單：");
    printUpperCase(names);
}

static void collectNameTo(List names) {
    Scanner console = new Scanner(System.in);
    while(true) {
        out.print("訪客名稱：");
        String name = console.nextLine();
        if(name.equals("quit")) {
            break;
        }
        names.add(name);
    }
}

static void printUpperCase(List names) {
    for(int i = 0; i < names.size(); i++) {
        String name = (String) names.get(i);
        out.println(name.toUpperCase());
    }
}
```

使用 Java SE 的 List 與 ArrayList

使用 get() 依索引取得收集之物件

具有索引的List

- **java.util.LinkedList**也實作了List介面，你可以將上面的範例中ArrayList換為LinkedList，而結果不變
- 那麼什麼時候該用ArrayList？何時該用LinkedList呢？

具有索引的List

- 陣列在記憶體中會是連續的線性空間，根據索引隨機存取時速度快
- 如果操作上有這類需求時，像是排序，就可使用ArrayList，可得到較好的速度表現
- 如果需要調整索引順序時，會有較差的表現
- 陣列的長度固定也是要考量的問題，為此，**ArrayList**有個可指定容量（Capacity）的建構式

具有索引的List

- LinkedList在實作List介面時，採用了鏈結（Link）結構

```
public class SimpleLinkedList {
    private class Node {
        Node(Object o) {
            this.o = o;
        }
        Object elem;
        Node next;
    }

    private Node first;

    public void add(Object elem) {
        Node node = new Node(elem);
        if(first == null) {
            first = node;
        }
        else {
            append(node);
        }
    }

    private void append(Node node) {
        Node last = first;
        while(last.next != null) {
            last = last.next;
        }
        last.next = node;
    }
```

❶ 將收集的物件用 Node 封裝

❷ 第一個節點

❸ 新增 Node 封裝物件，並由上一個 Node 的 next 參考

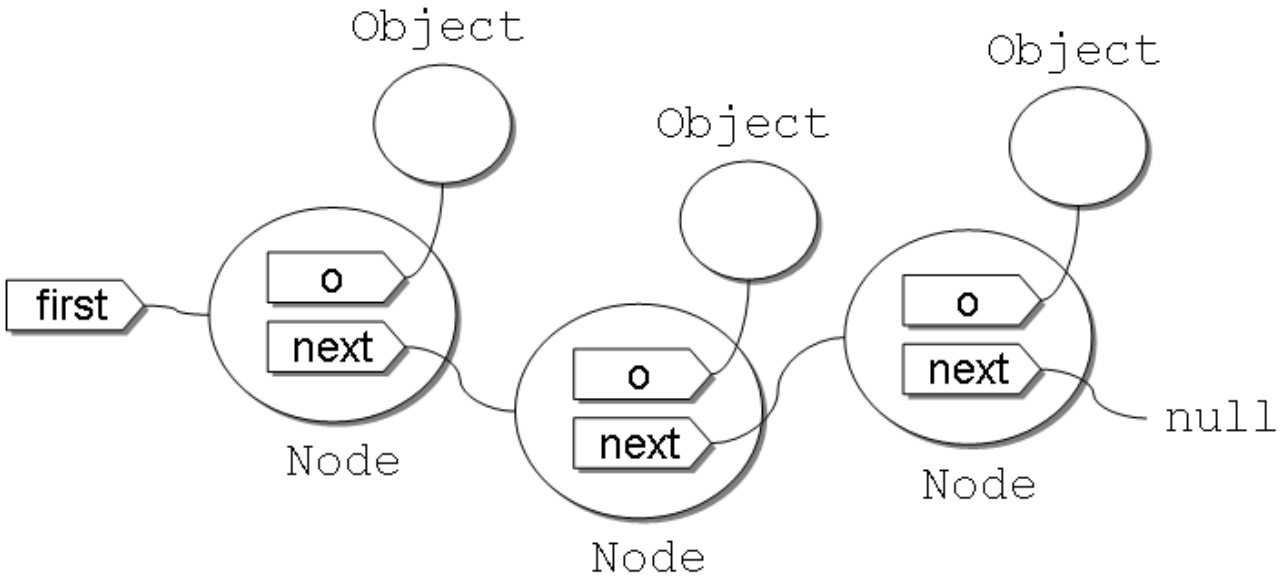
```
    public int size() {
        int count = 0;
        Node last = first;
        while(last != null) {
            last = last.next;
            count++;
        }
        return count;
    }

    public Object get(int index) {
        checkSize(index);
        return findElemOf(index);
    }

    private void checkSize(int index) throws IndexOutOfBoundsException {
        int size = size();
        if(index >= size) {
            throw new IndexOutOfBoundsException(
                    String.format("Index: %d, Size: %d", index, size));
        }
    }

    private Object findElemOf(int index) {
        int count = 0;
        Node last = first;
        while(count < index) {
            last = last.next;
            count++;
        }
        return last.elem;
    }
}
```

❹ 走訪所有 Node 並計數以取得長度

❺ 走訪所有 Node 並計數以取得對應索引物件

具有索引的List

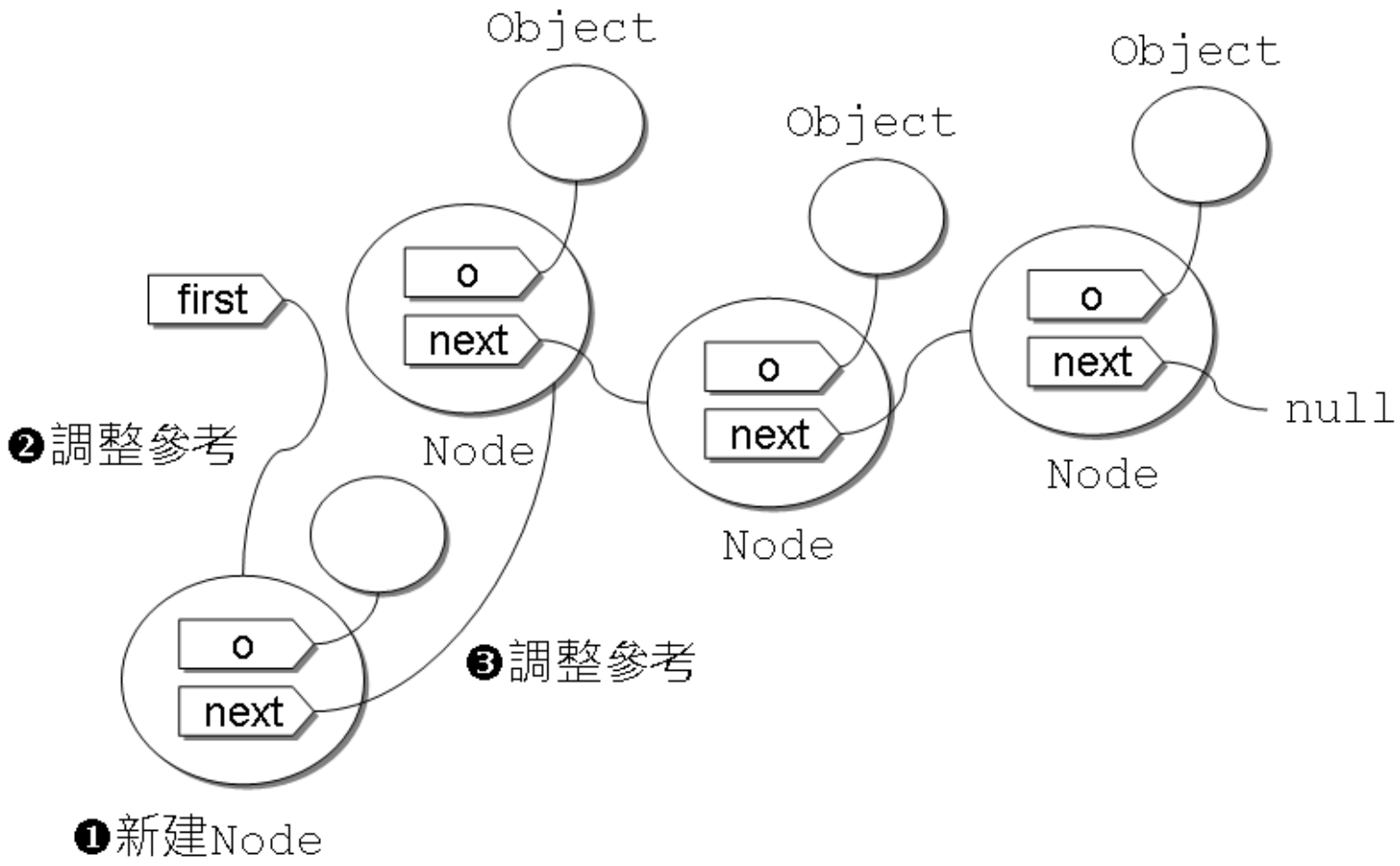
- 在SimpleLinkedList內部使用Node封裝新增的物件❶，每次add()新增物件之後，將會形成以下的鏈狀結構❸



具有索引的List

- 想要指定索引隨機存取物件時，鏈結方式都得使用從第一個元素開始查找下一個元素的方式，會比較沒有效率，像排序就不適合使用鏈結實作
- 鏈結的每個元素會參考下一個元素，這有利於調整索引順序

具有索引的List


Object
Object
Object
first
o
next
o
next
o
next
null
Node
Node
Node
❷調整參考
o
next
❸調整參考
❶新建Node

內容不重複的Set

- 若有一個字串，當中有許多的英文單字，你希望知道不重複的單字有幾個：

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);

    System.out.print("請輸入英文：");
    Set words = tokenSet(console.nextLine());
    System.out.printf("不重複單字有 %d 個：%s%n", words.size(), words);
}

static Set tokenSet(String line) {
    String[] tokens = line.split(" ");
    return new HashSet(Arrays.asList(tokens));
}
```

❶顯示收集的個數與字串

❷根據空白切割出字串

❸使用 HashSet 實作收集字串

內容不重複的Set

```
class Student {
    private String name;
    private String number;
    Student(String name, String number) {
        this.name = name;
        this.number = number;
    }

    @Override
    public String toString()  {
        return String.format("(%s, %s)", name, number);
    }
}
```

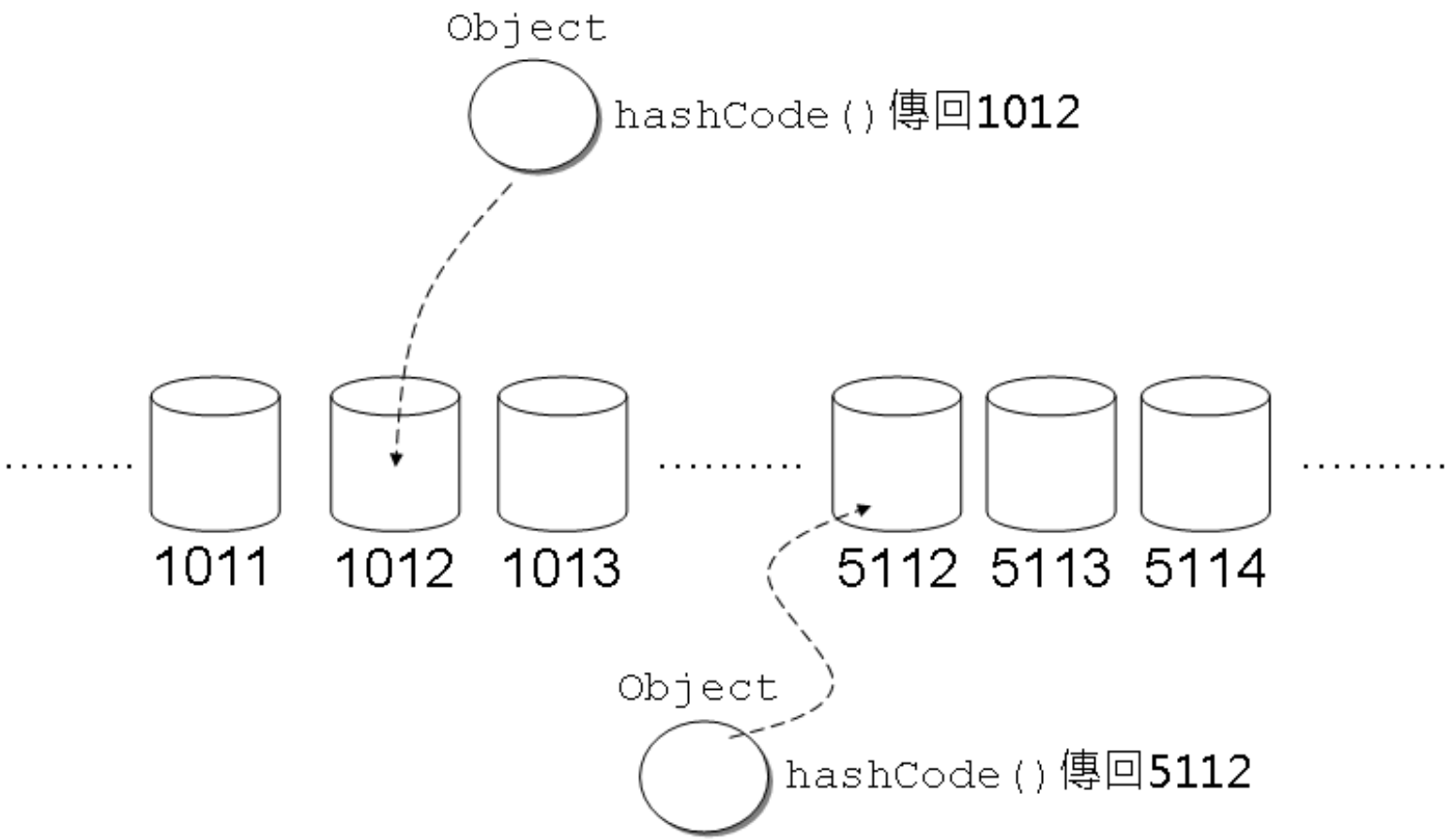
```
Set set = new HashSet();
set.add(new Student("Justin", "B835031"));
set.add(new Student("Monica", "B835032"));
set.add(new Student("Justin", "B835031"));
System.out.println(set);
```

```
[(Monica, B835032), (Justin, B835031), (Justin, B835031)]
```

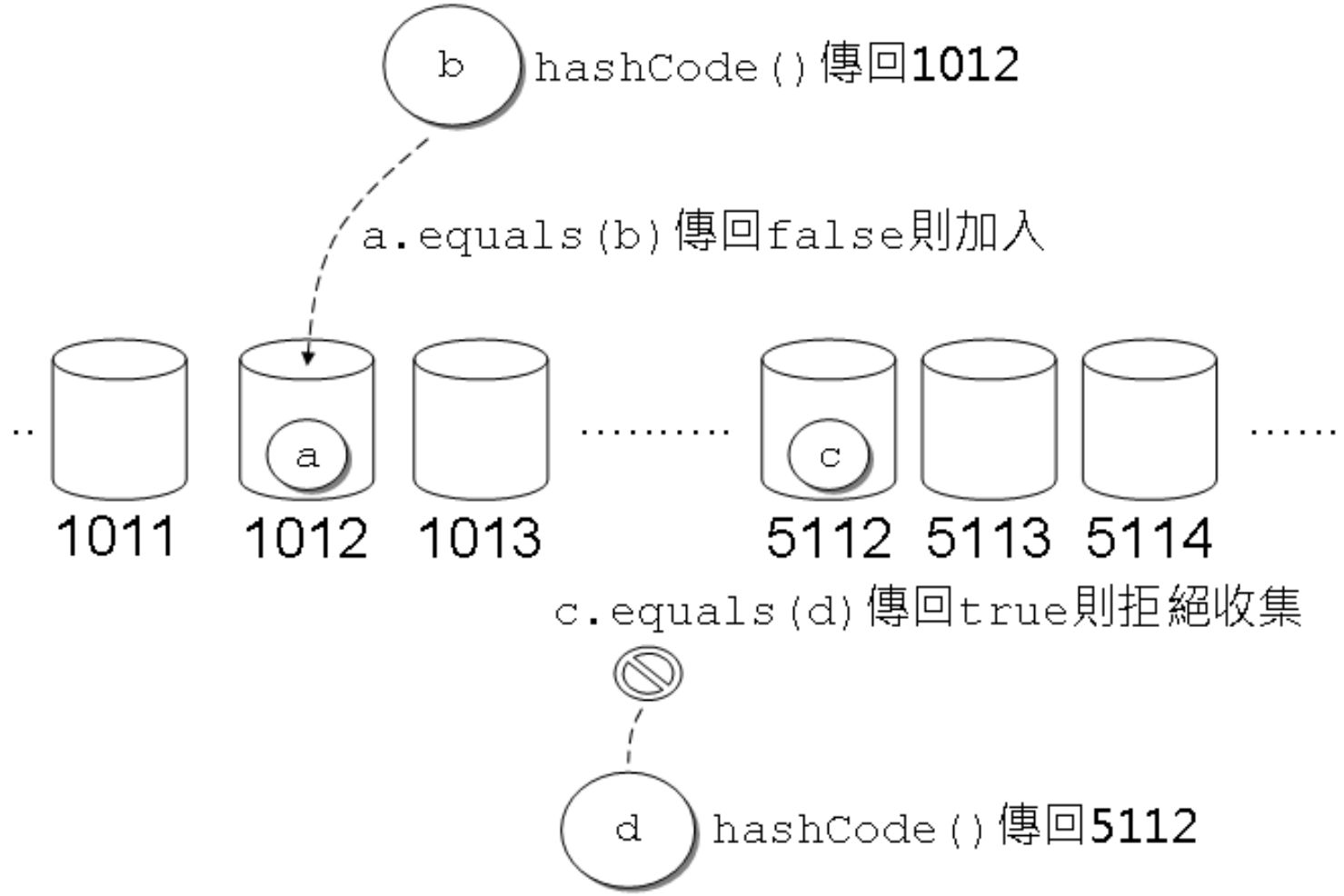
內容不重複的Set

- 你並沒有告訴Set，什麼樣的Student實例才算是重複...
- 以HashSet為例，會使用物件的hashCode()與equals()來判斷物件是否相同

內容不重複的Set


Object
hashCode()傳回1012
1011
1012
1013
5112
5113
5114
Object
hashCode()傳回5112

內容不重複的Set


b
hashCode()傳回1012
a.equals(b)傳回false則加入
a
c
1011
1012
1013
5112
5113
5114
c.equals(d)傳回true則拒絕收集
d
hashCode()傳回5112

內容不重複的Set

- Java中許多要判斷物件是否重複時，都會呼叫**hashCode()**與**equals()**方法
- 規格書中建議，兩個方法必須同時實作

```
public int hashCode() {
    // Objects 有 hash() 方法可以使用
    // 以下可以簡化為 return Objects.hash(name, number);
    int hash = 7;
    hash = 47 * hash + Objects.hashCode(this.name);
    hash = 47 * hash + Objects.hashCode(this.number);
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Student2 other = (Student2) obj;
    if (!Objects.equals(this.name, other.name)) {
        return false;
    }
    if (!Objects.equals(this.number, other.number)) {
        return false;
    }
    return true;
}
```

支援佇列操作的Queue

- Queue繼承自Collection，所以也具有Collection的add()、remove()、element()等方法
  - 操作失敗時會拋出例外
- Queue定義了自己的**offer()**、**poll()**與**peek()**等方法
  - 操作失敗時會傳回特定值

支援佇列操作的Queue

- offer()方法用來在佇列後端加入物件，成功會傳回true，失敗則傳回false
- poll()方法用來取出佇列前端物件，若佇列為空則傳回null
- peek()用來取得（但不取出）佇列前端物件，若佇列為空則傳回null

支援佇列操作的Queue

- **LinkedList**不僅實作了List介面，也實作了**Queue**的行為

```
interface Request {
    void execute();
}

public class RequestQueue {
    public static void main(String[] args) {
        Queue requests = new LinkedList();
        offerRequestTo(requests);
        process(requests);
    }

    static void offerRequestTo(Queue requests) {
        // 模擬將請求加入佇列
        for (int i = 1; i < 6; i++) {
            Request request = new Request() {
                public void execute() {
                    System.out.printf("處理資料 %f%n", Math.random());
                }
            };
            requests.offer(request);
        }
    }
    // 處理佇列中的請求
    static void process(Queue requests) {
        while(requests.peek() != null) {
            Request request = (Request) requests.poll();
            request.execute();
        }
    }
}
```

支援佇列操作的Queue

- 想對佇列的前端與尾端進行操作，在前端加入物件與取出物件，在尾端加入物件與取出物件，Queue的子介面**Deque**就定義了這類行為
- addFirst()、removeFirst()、getFirst()、addLast()、removeLast()、getLast()等方法，操作失敗時會拋出例外
- offerFirst()、pollFirst()、peekFirst()、offerLast()、pollLast()、peekLast()等方法，操作失敗時會傳回特定值

支援佇列操作的Queue

Queue 方法	Deque 等義方法
add()	addLast()
offer()	offerLast()
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

```
public class Stack {
    private Deque elems = new ArrayDeque();
    private int capacity;

    public Stack(int capacity) {
        this.capacity = capacity;
    }

    public boolean push(Object elem) {
        if(isFull()) {
            return false;
        }
        return elems.offerLast(elem);
    }

    private boolean isFull() {
        return elems.size() + 1 > capacity;
    }

    public Object pop() {
        return elems.pollLast();
    }

    public Object peek() {
        return elems.peekLast();
    }

    public int size() {
        return elems.size();
    }

    public static void main(String[] args) {
        Stack stack = new Stack(5);
        stack.push("Justin");
        stack.push("Monica");
        stack.push("Irene");
        out.println(stack.pop());
        out.println(stack.pop());
        out.println(stack.pop());
    }
}
```

使用泛型

- 在使用Collection收集物件時，由於事先不知道被收集物件之形態，因此內部實作時，都是使用Object來參考被收集之物件
- 取回物件時也是以Object型態傳回，原理可參考6.2.5自行實作的ArrayList，或9.1.2實作的SimpleLinkedList

使用泛型

- 若你想針對某類別定義的行為操作時，必須告訴編譯器，讓物件重新扮演該型態

```
List names = Arrays.asList("Justin", "Monica", "Irene") ;
String name = (String) words.get(0);
```

- 執行時期被收集的物件會失去形態資訊

使用泛型

- 實際上通常`Collection`中會收集同一種類型的物件
- 從JDK5之後，新增了泛型（Generics）語法，讓你在設計API時可以指定類別或方法支援泛型
- 使用API的客戶端在語法上會更為簡潔，並得到編譯時期檢查

GOTO

```
public class ArrayList<E> {
    private Object[] elems;
    private int next;

    public ArrayList(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList() {
        this(16);
    }

    public void add(E e) {
        if(next == elems.length) {
            elems = Arrays.copyOf(elems, elems.length * 2);
        }
        elems[next++] = e;
    }

    public E get(int index) {
        return (E) elems[index];
    }

    public int size() {
        return next;
    }
}
```

❶ 此類別支援泛型

❷ 加入的物件必須是客戶端宣告的 E 型態

❸ 取回物件以客戶端宣告的 E 型態傳回

技術手冊

使用泛型

- 使用泛型語法，會對設計API造成一些語法上的麻煩，但對客戶端會多一些友善

```
ArrayList<String> names = new ArrayList<String>();
names.add("Justin");
names.add("Monica");
String name1 = names.get(0);
String name2 = names.get(1);
```

使用泛型

- 宣告與建立物件時，可使用角括號告知編譯器，這個物件收集的都會是`String`，而取回之後也會是`String`
- 加入了不是`String`的東西會如何呢？

```
ArrayList<String> names
names.add("Justin");
names.add("Monica");
names.add(new Integer(10));
```

incompatible types: Integer cannot be converted to String

(Alt-Enter shows hints)

使用泛型

- Java的Collection API都支援泛型語法，若在API文件看到角括號，表示支援泛型語法

java.util

Interface Collection<E>

Type Parameters:

E - the type of elements in this collection

All Superinterfaces:

Iterable<E>

使用泛型

- 以使用java.util.List為例：

```
List<String> words = new LinkedList<String>();
words.add("one");
String word = words.get(0);
```

- 泛型語法有一部份是編譯器蜜糖（一部份是記錄於位元碼中的資訊）

```
LinkedList linkedlist = new LinkedList();
linkedlist.add("one");
String s = (String) linkedlist.get(0);
```

使用泛型

- 以下會編譯錯誤：

```
List<String> words = new LinkedList<String>();
words.add("one");
Integer number = words.get(0); // 編譯錯誤


List words = new LinkedList();
words.add("one");
Integer number = (String) words.get(0); // 編譯錯誤
```

使用泛型

- 若介面支援泛型，在實作時也會比較方便，例如：

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

class StringComparator2 implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return -s1.compareTo(s2);
    }
}
```

使用泛型

- 再來看一下以下程式片段：

```
List<String> words = new LinkedList<String>();
```

- 從JDK7之後有了點改善：

```
List<String> words = new LinkedList<>();
```

使用泛型

- 泛型也可以僅定義在方法上，最常見的是在靜態方法上定義泛型

```
public static Object elemOf(Object[] objs, int index) {
    return objs[index];
}
String arg = (String) elemOf(args, i);

public static <T> T elemOf(T[] objs, int index) {
    return objs[index];
}
String arg = elemOf(args, i);
```

簡介Lambda表示式

- 回顧一下9.1.4中的RequestQueue範例

```
Request request = new Request() {
    public void execute() {
        out.printf("處理資料 %f%n", Math.random());
    }
};
```

- 在JDK8中可以使用Lambda表示式

```
Request request = () -> out.printf("處理資料 %f%n", Math.random());
```

簡介Lambda表示式

- 相對於匿名類別語法來說，Lambda表示式的語法...
  - 省略了介面型態與方法名稱
  - ->左邊是參數列，而右邊是方法本體
  - 編譯器可以由`Request request`的宣告中得知語法上被省略的資訊

簡介Lambda表示式

- 如果有個介面宣告如下：

```
public interface IntegerFunction {
    Integer apply(Integer i);
}
```

- 使用匿名類別來實作

```
IntegerFunction doubleFunction = new IntegerFunction() {
    public Integer apply(Integer i) {
        return i * 2;
    }
}
```

簡介Lambda表示式

- 改用JDK8的Lambda表示式

```
IntegerFunction doubleFunction = (Integer i) -> i * 2;

IntegerFunction doubleFunction = (i) -> i * 2;

IntegerFunction doubleFunction = i -> i * 2;
```

簡介Lambda表示式

- 在使用Lambda表示式，編譯器在推斷型態時，還可以用泛型宣告的型態作為資訊來源

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

Comparator<String> byLength = new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
};
```

```
Comparator<String> byLength = (name1, name2) -> name1.length() - name2.length();
```

簡介Lambda表示式

- 來改寫一下9.1.4中的RequestQueue範例

```
static void offerRequestTo(Queue<Request2> requests) {
    // 模擬將請求加入佇列
    for (int i = 1; i < 6; i++) {
        requests.offer(
            () -> System.out.printf("處理資料 %f%n", Math.random())
        );
    }
}
```

簡介Lambda表示式

- 若流程較為複雜，無法於一行的Lambda表示式中寫完時，可以使用區塊{ }符號包括演算流程

```
Request request = () -> {
    out.printf("處理資料 %f%n", Math.random());
};

IntegerFunction doubleFunction = i -> {
    return i * 2;
}
```

Iterable與Iterator

- 如果要你寫個forEach()方法，可以顯示List收集的所有物件，也許你會這麼寫：

```
static void forEach(List list) {
    int size = list.size();
    for(int i = 0; i < size; i++) {
        out.println(list.get(i));
    }
}
```

Iterable與Iterator

- 如果要讓你寫個forEach()方法顯示Set收集的所有物件，你該怎麼寫呢？

```
static void forEach(Set set) {
    for(Object o : set.toArray()) {
        out.println(o);
    }
}
```

Iterable與Iterator

- 如果現在要讓你再實作一個forEach()方法，可以顯示Queue收集的所有物件，也許你會這麼寫：

```
static void forEach(Queue queue) {
    while(queue.peek() != null) {
        out.println(queue.poll());
    }
}
```

Iterable與Iterator

- 無論是List、Set或Queue，都會有個**iterator()**方法
  - 在JDK1.4之前，是定義在**Collection**介面中
  - 在JDK5之後，原先定義在Collection中的iterator()方法，提昇至新的java.util.Iterable父介面

Iterable與Iterator

- iterator()方法會傳回**java.util.Iterator**介面的實作物件
  - 可以使用Iterator的**hasNext()**看看有無下一個物件，若有的話，再使用**next()**取得下一個物件

```
static void forEach(Collection collection) {
    Iterator iterator = collection.iterator();
    while(iterator.hasNext()) {
        out.println(iterator.next());
    }
}
```

Iterable與Iterator

- 在JDK5之後，你可以使用以下的forEach()方法顯示收集的所有物件：

```
static void forEach(Iterable iterable) {
    Iterator iterator = iterable.iterator();
    while(iterator.hasNext()) {
        out.println(iterator.next());
    }
}
```

Iterable與Iterator

- 在JDK5之後有了增強式**for**迴圈
  - 運用在陣列上
  - 運用在實作**Iterable**介面的物件上

```
static void forEach(Iterable iterable) {
    for(Object o : iterable) {
        System.out.println(o);
    }
}

List names = Arrays.asList("Justin", "Monica", "Irene"); ❶
forEach(names); ❷
forEach(new HashSet(names)); ❸
forEach(new ArrayDeque(names)); ❹
```

Iterable與Iterator

- 增強式**for**迴圈是編譯器蜜糖，當運用在Iterable物件時，會展開為：

```
private static void forEach(Iterable iterable) {
    Object o;
    for(Iterator i$ = iterable.iterator();
                i$.hasNext();
                System.out.println(o)) {
        o = i$.next();
    }
}
```

Iterable與Iterator

- 如果使用JDK8，想要迭代物件還有新的選擇，Iterable上新增了forEach()方法

```
List<String> names = Arrays.asList("Justin", "Monica", "Irene");
names.forEach(name -> out.println(name));
new HashSet(names).forEach(name -> out.println(name));
new ArrayDeque(names).forEach(name -> out.println(name));
```

Comparable與Comparator

- java.util.Collections提供有sort()方法，由於必須有索引才能進行排序，因此Collections的sort()方法接受List實作物件

```
List numbers = Arrays.asList(10, 2, 3, 1, 9, 15, 4);
Collections.sort(numbers);
System.out.println(numbers);
```

```
[1, 2, 3, 4, 9, 10, 15]
```

```
class Account {
    private String name;
    private String number;
    private int balance;

    Account(String name, String number, int balance) {
        this.name = name;
        this.number = number;
        this.balance = balance;
    }

    @Override
    public String toString() {
        return String.format("Account(%s, %s, %d)", name, number, balance);
    }
}

public class Sort2 {
    public static void main(String[] args) {
```

```
Exception in thread "main" java.lang.ClassCastException: cc.openhome.Account
cannot be cast to java.lang.Comparable
...略
```

```
        );
        Collections.sort(accounts);
        System.out.println(accounts);
    }
}
```

Comparable與Comparator

- Collections的sort()方法要求被排序的物件，必須實作**java.lang.Comparable**介面

Comparable與Comparator

```
class Account2 implements Comparable<Account2> {
    private String name;
    private String number;
    private int balance;

    Account2(String name, String number, int balance) {
        this.name = name;
        this.number = number;
        this.balance = balance;
    }

    @Override
    public String toString() {
        return String.format("Account2(%s, %s, %d)", name, number, balance);
    }

    @Override
    public int compareTo(Account2 other) {
        return this.balance - other.balance;
    }
}
```

Comparable與Comparator

- 為何先前的Sort類別中，可以直接對Integer進行排序呢？

```
public final class Integer
extends Number
implements Comparable<Integer>
```

Comparable與Comparator

- 如果你的物件無法實作Comparable呢？

```
List words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
Collections.sort(words);
System.out.println(words);
```

```
[A, B, F, M, O, W, X]
```

Comparable與Comparator

- Collections的sort()方法有另一個重載版本，可接受**java.util.Comparator**介面的實作物件

```
class StringComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return -s1.compareTo(s2);
    }
}
```

```
[X, W, O, M, F, B, A]
```

```
pub
    public static void main(String[] args) {
        List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
        Collections.sort(words, new StringComparator());
        System.out.println(words);
    }
}
```

Comparable與Comparator

- 如果想針對陣列進行排序，可以使用**java.util.Arrays**的**sort()**方法
  - 該方法針對物件排序時有兩個版本，一個是你收集在陣列中的物件必須是Comparable（否則會拋出ClassCastException），另一個版本則可以傳入Comparator指定排序方式

Comparable與Comparator

- Comparator介面需要實作的只有一個compare()方法

```
List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
Collections.sort(words, (s1, s2) -> -s1.compareTo(s2));
```

- JDK8在List上增加了sort()方法，可接受Comparator實例

```
List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
words.sort((s1, s2) -> -s1.compareTo(s2));
```

Comparable與Comparator

- 如果有個List中某些索引處包括null，現在你打算讓那些null排在最前頭，之後依字串的長度由大到小排序

Comparable與Comparator

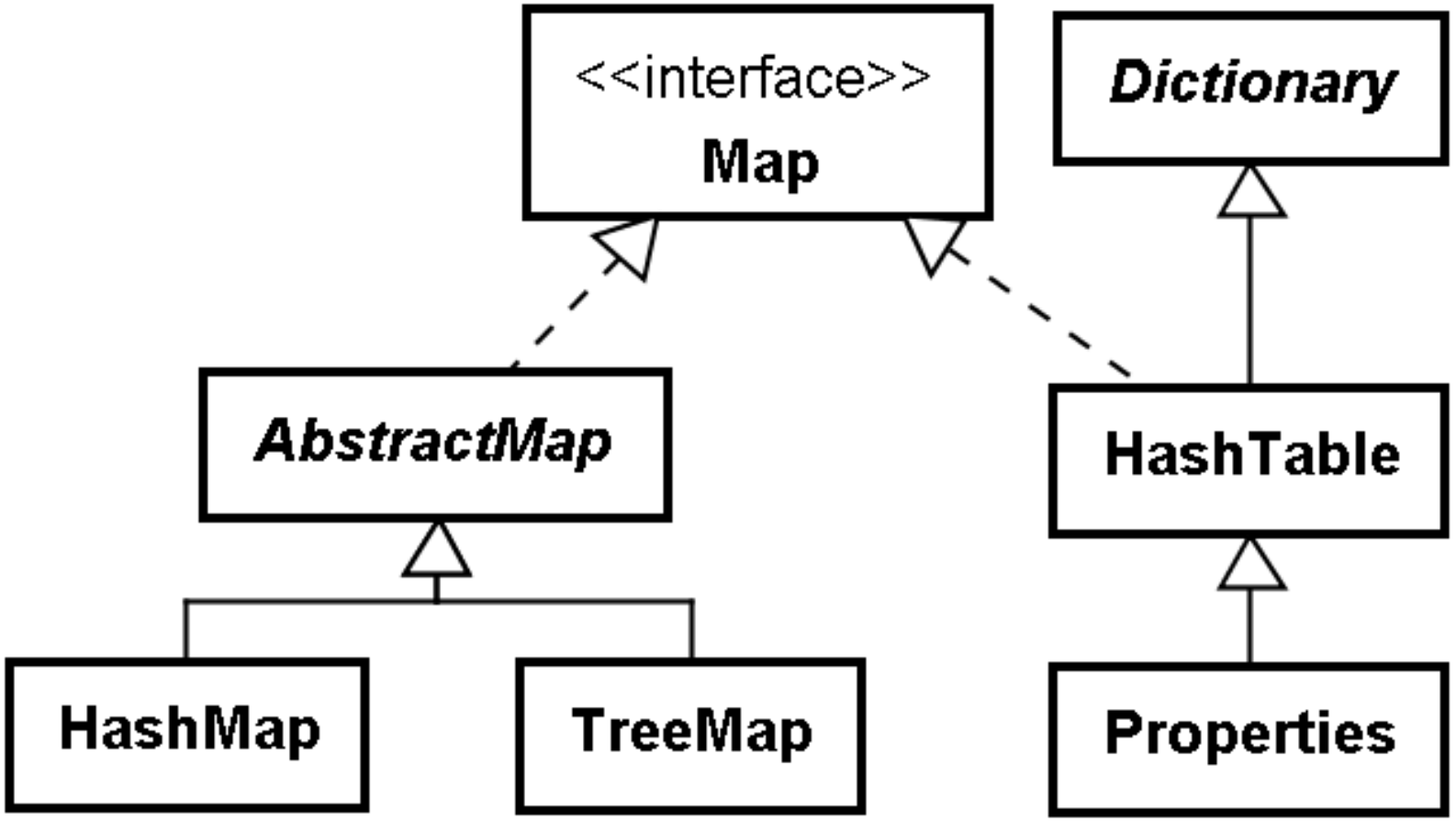
```
public class StrLengthInverseNullFirstComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        if(s1 == s2) {
            return 0;
        }
        if(s1 == null) {
            return -1;
        }
        if(s2 == null) {
            return 1;
        }
        if(s1.length() == s2.length()) {
            return 0;
        }
        if(s1.length() > s2.length()) {
            return -1;
        }
        return 1;
    }
}
```

Comparable與Comparator

- JDK8為排序加入了一些高階語義API

```
List words = Arrays.asList(
        "B", "X", "A", "M", null ,"F", "W", "O", null);
words.sort(nullsFirst(reverseOrder()));
System.out.println(words);
```

常用Map實作類別


<<interface>>
Map
Dictionary
AbstractMap
HashTable
HashMap
TreeMap
Properties

常用Map實作類別

```
Map<String, String> messages = new HashMap<>();
messages.put("Justin", "Hello！Justin 的訊息！");
messages.put("Monica", "給 Monica 的悄悄話！");
messages.put("Irene", "Irene 的可愛貓喵喵叫！");

Scanner console = new Scanner(System.in);
out.print("取得誰的訊息：");
String message = messages.get(console.nextLine());
out.println(message);
out.println(messages);
```

❶ 以泛型語法指定鍵值型態

❷ 建立鍵值對應

❸ 指定鍵取回值

常用Map實作類別

- HashMap中建立鍵值對應之後，鍵是無序的
- 如果想讓鍵是有序的，則可以使用TreeMap
  - 鍵的部份將會排序，條件是作為鍵的物件必須實作**Comparable**介面，或者是在建構**TreeMap**時指定實作**Comparator**介面的物件

```
Map<String, String> messages = new TreeMap<>();
messages.put("Justin", "Hello！Justin 的訊息！");
messages.put("Monica", "給 Monica 的悄悄話！");
messages.put("Irene", "Irene 的可愛貓喵喵叫！");
System.out.println(messages);
```

常用Map實作類別

- 想看到相反的排序結果，那麼可以如下實作Comparator：

```
Map<String, String> messages =
        new TreeMap<>((s1, s2) -> -s1.compareTo(s2));

messages.put("Justin", "Hello！Justin 的訊息！");
messages.put("Monica", "給 Monica 的悄悄話！");
messages.put("Irene", "Irene 的可愛貓喵喵叫！");
System.out.println(messages);
```

常用Map實作類別

- Properties的**setProperty()**指定字串型態的鍵值，**getProperty()**指定字串型態的鍵，取回字串型態的值

```
Properties props = new Properties();
props.setProperty("username", "justin");
props.setProperty("password", "123456");
System.out.println(props.getProperty("username"));
System.out.println(props.getProperty("password"));
```

常用Map實作類別

- Properties也可以從檔案中讀取屬性

Map person.properties

```
# 使用者名稱與密碼
cc.openhome.username=justin
cc.openhome.password=123456
```

```
Properties props = new Properties();
props.load(new FileInputStream(args[0]));
System.out.println(props.getProperty("cc.openhome.username"));
System.out.println(props.getProperty("cc.openhome.password"));
```

常用Map實作類別

- 也可以使用**loadFromXML()**方法載入.xml檔案

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment></comment>
    <entry key="cc.openhome.username">justin</entry>
    <entry key="cc.openhome.username">123456</entry>
</properties>
```

常用Map實作類別

- 在使用java指令啟動JVM時，可以使用-D指定系統屬性：

```
> java -Dusername=justin -Dpassword=123456 LoadSystemProps
```

```
Properties props = System.getProperties();
System.out.println(props.getProperty("username"));
System.out.println(props.getProperty("password"));
```

常用Map實作類別

- System.getProperties()取回的Properties實例中，包括了許多預置屬性

getProperties

```
public static Properties getProperties()
```

Determines the current system properties.

First, if there is a security manager, its `checkPropertiesAccess` method is called with no arguments. This may result in a security exception.

The current set of system properties for use by the `getProperty(String)` method is returned as a `Properties` object. If there is no current set of system properties, a set of system properties is first created and initialized. This set of system properties always includes values for the following keys:

Key	Description of Associated Value
java.version	Java Runtime Environment version
java.vendor	Java Runtime Environment vendor
java.vendor.url	Java vendor URL
java.home	Java installation directory
java.vm.specification.version	Java Virtual Machine specification version
java.vm.specification.vendor	Java Virtual Machine specification vendor
java.vm.specification.name	Java Virtual Machine specification name
java.vm.version	Java Virtual Machine implementation version
java.vm.vendor	Java Virtual Machine implementation vendor
java.vm.name	Java Virtual Machine implementation name
java.specification.version	Java Runtime Environment specification version
java.specification.vendor	Java Runtime Environment specification vendor

走訪Map鍵值

```
Map<String, String> map = new HashMap<>();
map.put("one", "一");
map.put("two", "二");
map.put("three", "三");

out.println("顯示鍵");
// keySet()傳回 Set
map.keySet().forEach(key -> out.println(key));

out.println("顯示值");
// values()傳回 Collection
map.values().forEach(key -> out.println(key));
```

走訪Map鍵值

- 如果想同時取得Map的鍵與值，可以使用**entrySet()**方法

```
public static void main(String[] args) {
    Map<String, String> map = new TreeMap<>();
    map.put("one", "一");
    map.put("two", "二");
    map.put("three", "三");
    foreach(map.entrySet());
}

public static void foreach(
                    Iterable<Map.Entry<String, String>> iterable) {
    for(Map.Entry<String, String> entry: iterable) {
        System.out.printf("(鍵 %s, 值 %s)%n",
                entry.getKey(), entry.getValue());
    }
}
```

走訪Map鍵值

- 泛型語法用到某個程度時，老實說可讀性並不好 ...
- 撰寫程式還是得兼顧可讀性

```
Map map = new TreeMap();
map.put("one", "一");
map.put("two", "二");
map.put("three", "三");
map.forEach(
    (key, value) -> System.out.printf("(鍵 %s, 值 %s)%n", key, value)
);
```