

# 工具類別(2)

TYIC 桃高資訊社

# 列舉

列舉(**enumerate**)，顧名思義，就是把東西列出來

在 **Java** 中，**列舉**就是一個特殊的**類別**：

1. **列舉類別**為**不可繼承類別**，也不可以**繼承類別**，但可**實作介面**
2. 其中的**常數**就是**公開靜態不可變欄位**，為該**列舉類別**的**實例**
3. **列舉類別**的**建構子**為 **private**，外界不可**實例化**

```
修飾子 enum 列舉類別名稱 {常數1, 常數2, ..., 常數n} java
```

```
修飾子 enum 列舉類別名稱 {  
    常數1(引數...), 常數2(引數...), ..., 常數n(引數...);
```

欄位...

方法...

建構子...

類別...

```
}
```

java

# 列舉

列舉若實作介面  
可以等到創建實例  
才覆寫方法

列舉類別的  
公開靜態方法

**Role values()**

可以返回該

列舉類別的常數

組成的陣列

```
public class Main {
    public static void main(String[] args) {
        for (Role role : Role.values()) {
            new Person(role).printInfo().eat();
        }
    }
}

enum Role implements Eat {
    WORKER("上班族") {
        @Override
        public void eat() {
            System.out.println("吃土");
        }
    }, BABY("嬰兒") {
        @Override
        public void eat() {
            System.out.println("喝奶");
        }
    }, STUDENT("學生") {
        @Override
        public void eat() {
            System.out.println("叫外送");
        }
    };

    final String description;

    Role(String description) {
        this.description = description;
    }
}
```

```
interface Eat {
    void eat();
}
```

```
class Person {
    Role role;
```

```
    Person(Role role) {
        this.role = role;
    }
```

```
    Person printInfo() {
        System.out.println(
            role.description +
            " : " + switch (role) {
                case WORKER -> "上班";
                case BABY -> "哭";
                case STUDENT -> "上課";
            });
        return this;
    }
```

```
    void eat() {
        role.eat();
    }
}
```



上班族：上班  
吃土  
嬰兒：哭  
喝奶  
學生：上課  
叫外送

output

java

# Comparable 與 Comparator

**Comparable<T>** 為一介面，表示可比較的  
該介面定義了 **int compareTo(T o)** 方法  
該方法回傳值為負整數時，表示 **o1** 小於 **o2**  
回傳值為 **0** 時，表示 **o1** 等於 **o2**  
回傳值為正整數時，表示 **o1** 大於 **o2**

**Comparator<T>** 為一函式介面，表示比較器  
該函式介面定義了 **int compare(T o1, T o2)** 方法  
該方法回傳值代表意義與上方的 **int compareTo(T o)** 方法相同

# 陣列排序

之前有介紹過，要排序陣列可以使用

`java.util.Arrays` 的公開靜態方法 `void sort(array)`

但使用這個方法的前提是 `array` 須為基本資料型別陣列

或是 `array` 的所有元素皆實作 `Comparable<T>` 介面

若想對元素沒有實作 `Comparable<T>` 介面的陣列排序

或是不想依照元素實作 `Comparable<T>` 介面的方法排序

可以使用公開靜態方法來進行自定義排序：

```
<T> void sort(T[] a, Comparator<? super T> c)
```

在使用二分搜尋法搜尋時須使用同個比較器：

```
<T> void binarySearch(T[] a, T key, Comparator<? super T> c)
```

# 陣列排序

```
import java.util.Arrays;

public class Main8 {
    public static void main(String[] args) {
        Person[] people = new Person[]{
            new Person(50, "善依純"),
            new Person(10, "蕭亞瑄"),
            new Person(40, "郭靖"),
            new Person(30, "陳生"),
            new Person(20, "徐懷豫")
        };
        Arrays.sort(people);
        System.out.println(Arrays.toString(people));
        Arrays.sort(people, (person1, person2) ->
            person2.age - person1.age);
        System.out.println(Arrays.toString(people));
    }
}
```

```
class Person implements Comparable<Person> {
    int age;
    String name;

    Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "姓名:%s 年齡:%d".formatted(name, age);
    }

    @Override
    public int compareTo(Person o) {
        return age - o.age;
    }
}
```

java



```
[姓名：蕭亞瑄 年齡：10, 姓名：徐懷豫 年齡：20, 姓名：陳生 年齡：30, 姓名：郭靖 年齡：40, 姓名：善依純 年齡：50]
[姓名：善依純 年齡：50, 姓名：郭靖 年齡：40, 姓名：陳生 年齡：30, 姓名：徐懷豫 年齡：20, 姓名：蕭亞瑄 年齡：10] output
```

# 函式介面

Java 還提供許多定義好的函式介面可以使用  
大部分位於 `java.util.function` 套件下，常見的有：

- `Supplier<T>` 生產者，不接收、返回 `T`
- `Consumer<T>` 消費者，接收 `T`、不返回
- `Function<T, R>` 函式，接收 `T`、返回 `R`
- `Predicate<T>` 述詞，接收 `T`、返回 `Boolean`
- `java.lang.Runnable` 可執行函式，不接收、不返回

以及可以接收兩個值的變種，如：`BiConsumer<T, U>`  
還有許多不使用泛型而是固定型別的變種，如 `IntConsumer`

# Optional

`java.util.Optional<T>` 是專門用來進行空值處理的類別，下方為部分公開方法：

```
static <T> Optional<T> empty()、static <T> Optional<T> of(T value)
static <T> Optional<T> ofNullable(T value)
boolean isPresent()、boolean isEmpty()
Optional<T> filter(Predicate<? super T> predicate)
T get()、T orElse(T other)、T orElseGet(Supplier<? extends T> supplier)
void ifPresent(Consumer<? super T> action)
void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)
<U> Optional<U> map(Function<? super T, ? extends U> mapper)
```



# Optional

```
import java.util.Optional;

public class Main {
    public static String[] stringArr;

    public static void main(String[] args) {
        Util.printElement(stringArr, 0, "空");
        stringArr = new String[5];
        stringArr[0] = "烏黑的髮尾";
        stringArr[1] = "盤成一個圈";
        Util.printElement(stringArr, 0, "空");
        Util.printElement(stringArr, 2, "空");
    }
}

abstract class Util {
    public static Optional<String> getElement(String[] arr, int index) {
        if (arr == null || index < 0 || index >= arr.length) return Optional.empty();
        return Optional.of(arr[index] != null ? arr[index] : "(null)");
    }

    public static void printElement(String[] arr, int index, String defaultPrint) {
        Optional<String> string = getElement(arr, index);
        System.out.println(string.orElse(defaultPrint));
    }
}
```

空  
烏黑的髮尾  
(null)

output



java

# 集合框架

集合框架(collection framework) 的功能是用來蒐集資料  
主要分為兩個介面：

`java.util.Collection<E>` 和 `java.util.Map<K, V>`

`Collection<E>` 代表一些相同型別的物件放在一起

`Map<K, V>` 代表一些相同型別的鍵(key)

各自映射到一個相同型別的值(value)

其中鍵不能重複，稱為鍵值映射(key-value mapping)，

每組鍵和值稱為「鍵值對(key-value pair、map entry)」

# 集合框架

集合框架與資料結構(data structure)高度相關

但資料結構是個理論

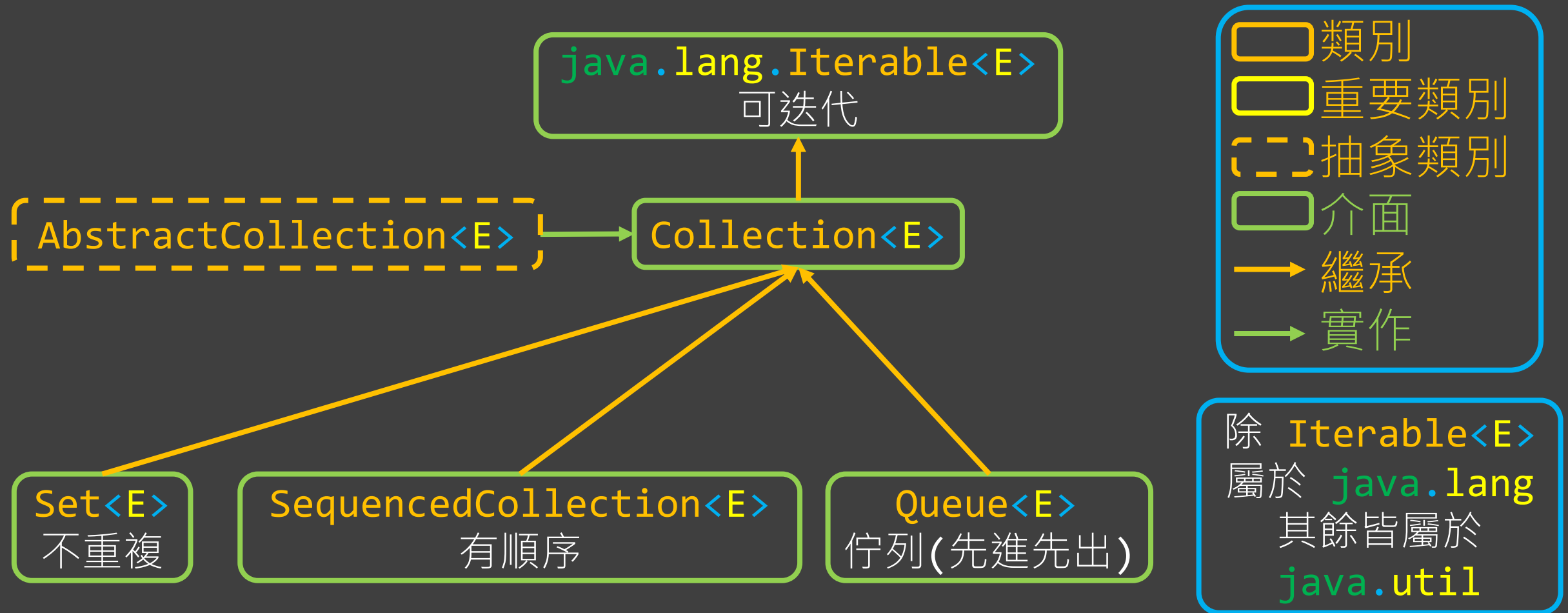
而集合框架則是 Java 中對於資料結構的實作

故在各個程式語言中，對於資料結構的實作可能有些許差異

但資料結構理論在任何程式語言皆是恆定的

在儲存資料時，應該要仔細思考當前情境適合使用哪種資料結構  
避免造成空間與時間的大量損失

# Collection



# Iterable 與 Iterator

`Collection<E>` 介面繼承了 `java.lang.Iterable<E>` 介面  
該介面表示可迭代的，其中有兩個公開動態方法：

`Iterator<E> iterator()`

`void forEach(Consumer<? super E> action)`

`Iterator<E>` 介面代表迭代器，其中有四個公開動態方法：

`void forEachRemaining(Consumer<? super E> action)`

`boolean hasNext()`、`E next()`、`void remove()`

# Collection

下方為 `Collection<E>` 介面部分公開動態方法：

`int size()`、`boolean isEmpty()`、`void clear()`

`boolean add(E e)`、`boolean addAll(Collection<? extends E> c)`

`boolean remove(Object o)`、`boolean removeAll(Collection<?> c)`

`boolean removeIf(Predicate<? super E> filter)`

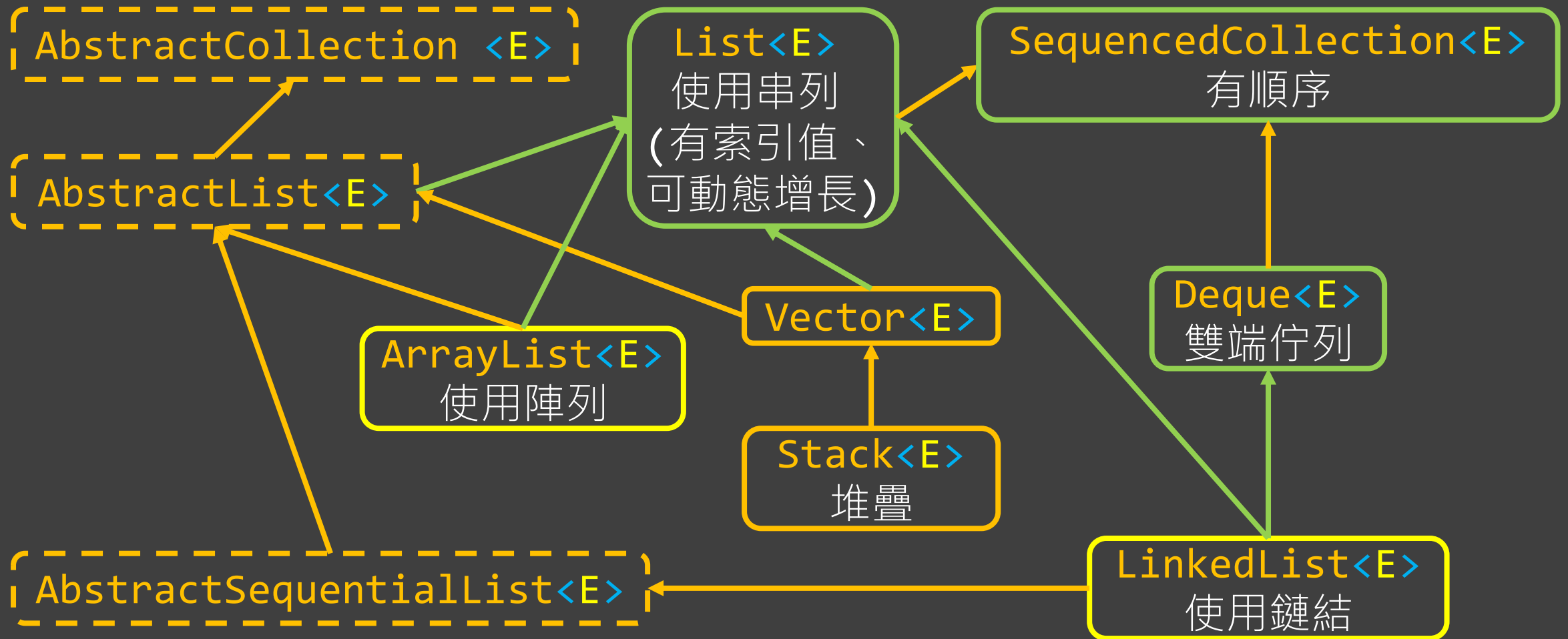
`boolean contains(Object o)`

`boolean containsAll(Collection<?> c)`、`<T> T[] toArray(T[] a)`

`Collection<E>` 的所有實作類別都有一個建構子

其唯一參數為 `Collection<? extends E>`

# List



# SequencedCollection

**SequencedCollection<E>** 介面在 Java 21 才出現

其實作類別的元素是有順序的

該介面定義了以下的公開動態方法：

**SequencedCollection<E>** reversed()

void add(int index, E element)

void addFirst(E e)、void addLast(E e)

E getFirst()、E getLast()

E removeFirst()、E removeLast()



# List

**List<E>** 介面的實作類別可以像陣列一樣儲存元素  
但是儲存容量可以動態增長，也就是說元素的數量可以不固定  
常被稱為「串列(list)」

下方為該介面的部分公開動態方法：

```
int indexOf(Object o)、int lastIndexOf(Object o)
```

```
void sort(Comparator<? super E> c)
```

```
List<E> subList(int fromIndex, int toIndex)
```

該介面還有一個靜態方法 **<E> List<E> of(E... elements)**

該介面最重要的實作類別是 **ArrayList<E>** 和 **LinkedList<E>**

# ArrayList

**ArrayList<E>** 類別內部使用陣列去實作串列

但陣列一旦創建就無法再更改長度

顯然與剛剛所說的串列儲存容量可動態增長有明顯的衝突

事實上，**ArrayList<E>** 內部的運作原理

就是在原本的陣列裝滿元素後

再創建容量變為 1.5 倍的新陣列

然後將舊陣列的元素複製到新陣列，並捨棄舊陣列

在 Java 中，**ArrayList<E>** 常常比陣列使用更多

# ArrayList

[姓名：陳生 年齡：20, 姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20]

0

2

[姓名：陳生 年齡：20, 姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20, 姓名：鄭宜 年齡：30]

[姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20, 姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20] output

```
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class Main {
    public static void main(String[] args) {
        ArrayList<Person> arrayList = new ArrayList<>(List.of(
            new Person(30, "鄭宜"),
            new Person(20, "陳生")
        ));
        arrayList.addFirst(new Person(20, "陳生"));
        System.out.println(arrayList);
        System.out.println(
            arrayList.indexOf(new Person(20, "陳生"))
        );
        System.out.println(
            arrayList.lastIndexOf(new Person(20, "陳生"))
        );
        arrayList.addLast(new Person(30, "鄭宜"));
        // 與 arrayList.add(new Person(30, "鄭宜")) 等價
        System.out.println(arrayList);
        System.out.println(arrayList.reversed());
    }
}
```


```
class Person {
    protected int age;
    String name;

    Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "姓名：%s 年齡：%d".formatted(name, age);
    }

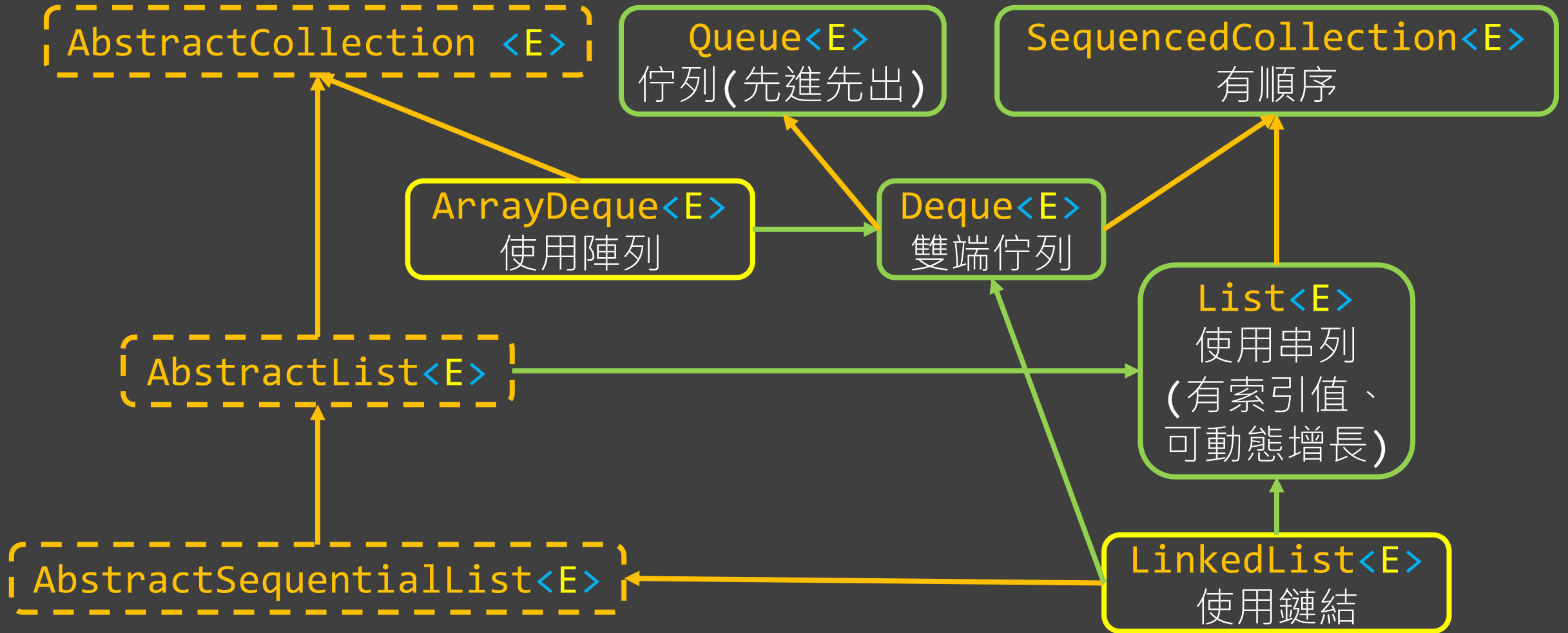
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(age, name);
    }
}
```



java

# Queue



# Queue

**Queue<E>** 介面的實作類別表示元素從尾部(**tail**)進入並從頭部(**head**)出來，常被稱為「佇列(**queue**)」

佇列的元素為先進先出(**FIFO**，**first-in-first-out**)

該介面定義了以下的公開動態方法：

**boolean offer(E e)** (將元素插入佇列尾部)

返回型別為 **E** 且無參數：**element**(獲取佇列頭部元素)

**peek**(獲取佇列頭部元素)、**poll**(刪除佇列頭部元素)

其中 **add** 與 **offer**、**element** 與 **peek**、**remove** 與 **poll** 三組方法的差異在於：前者在佇列為空時會報錯，後者則不會

該介面最重要的實作類別是 **ArrayDeque<E>** 和 **LinkedList<E>**

# Deque 與 Stack

**Deque<E>** 介面的實作類別表示元素從頭部或尾部進入並從頭部或尾部出來，常被稱為「雙端佇列(**deque**)」  
雙端佇列具有佇列和堆疊(**stack**)的性質

堆疊的元素只能從頭部進出

為後進先出(**LIFO**，**last-in-first-out**)

在 **Java** 中，並沒有堆疊介面，只有 **Stack<E>** 類別

但若要使用堆疊，推薦使用 **ArrayDeque<E>** 取代 **Stack<E>**

因為其和父類別 **Vector<E>** 是 **Java** 濫用繼承的失敗產物

# 佇列、雙端佇列、堆疊

尾部(tail)

佇列

頭部(head)



堆疊



雙端佇列



# Deque

**Deque<E>** 介面定義了以下的公開動態方法：

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

**boolean** `offerFirst(E e)`、**boolean** `offerLast(E e)`

**E** `peekFirst()`、**E** `peekLast()`、**E** `pollFirst()`、**E** `pollLast()`

堆疊操作：

**E** `pop()`

(從雙端佇列頭部彈出(**pop**)元素，即刪除頭部元素，等價於 `removeFirst`)

**boolean** `push(E e)`

(從雙端佇列頭部推入(**push**)元素，即將元素插入頭部，等價於 `addFirst`)



# ArrayDeque

**ArrayDeque<E>** 類別內部使用陣列去實作佇列  
與 **ArrayList<E>** 類似

**ArrayDeque<E>** 在陣列長度不足時會創建新陣列  
並將舊陣列內的元素複製到新陣列

在舊陣列長度小於 64 時

新陣列長度為舊陣列的 2 倍多 2

否則，新陣列長度為舊陣列的 1.5 倍

```

[姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20]
[姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20, 姓名：陳生 年齡：20]
[姓名：鄭宜 年齡：30, 姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20, 姓名：陳生 年齡：20]
姓名：鄭宜 年齡：30
[姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20, 姓名：陳生 年齡：20]
姓名：陳生 年齡：20
[姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20]
姓名：陳生 年齡：20
姓名：鄭宜 年齡：30

```

output

```

import java.util.ArrayDeque;
import java.util.List;
import java.util.Objects;

public class Main {
    public static void main(String[] args) {
        ArrayDeque<Person> arrayDeque = new ArrayDeque<>(List.of(
            new Person(30, "鄭宜"),
            new Person(20, "陳生")
        ));
        System.out.println(arrayDeque);
        arrayDeque.offerLast(new Person(20, "陳生"));
        System.out.println(arrayDeque);
        arrayDeque.offerFirst(new Person(30, "鄭宜"));
        System.out.println(arrayDeque);
        System.out.println(arrayDeque.pollFirst());
        System.out.println(arrayDeque);
        System.out.println(arrayDeque.pollLast());
        System.out.println(arrayDeque);
        for (var it = arrayDeque.descendingIterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
    }
}

```

# ArrayDeque



```

class Person {
    protected int age;
    String name;

    Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "姓名：%s 年齡：%d".formatted(name, age);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(age, name);
    }
}

```

java

# LinkedList

**LinkedList<E>** 類別內部使用雙向鏈結去實作串列和佇列  
常被稱為「鏈結串列(linked list)」

鏈結是指使用指標來紀錄上一個和下一個元素的記憶體位址  
與 **ArrayList<E>** 或 **ArrayDeque<E>** 使用陣列不同

但使用方法與 **ArrayList<E>** 或 **ArrayDeque<E>** 完全相同  
只紀錄上一個或下一個元素的串列

稱為「單向鏈結串列(singly linked list)」

同時紀錄上一個和下一個元素的串列

稱為「雙向鏈結串列(doubly linked list)」

# 陣列與鏈結串列

串列的好處在於增刪元素比較快，但壞處就是存取元素慢  
在插入元素時

陣列須將插入目標索引值的元素和所有後方的元素向後移動  
而鏈結串列只需要修改鏈結就可以了

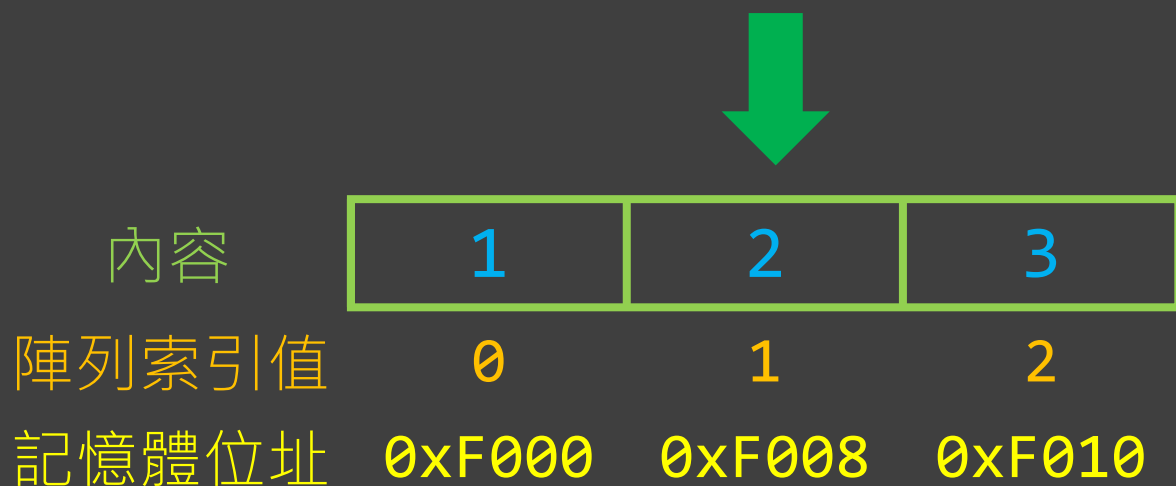
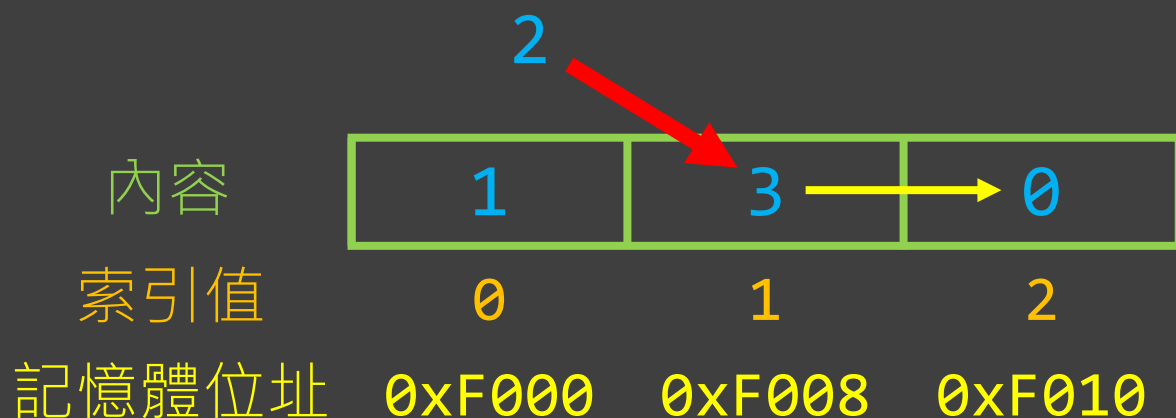
但在存取元素時

陣列只需進行記憶體位址的簡單數學運算

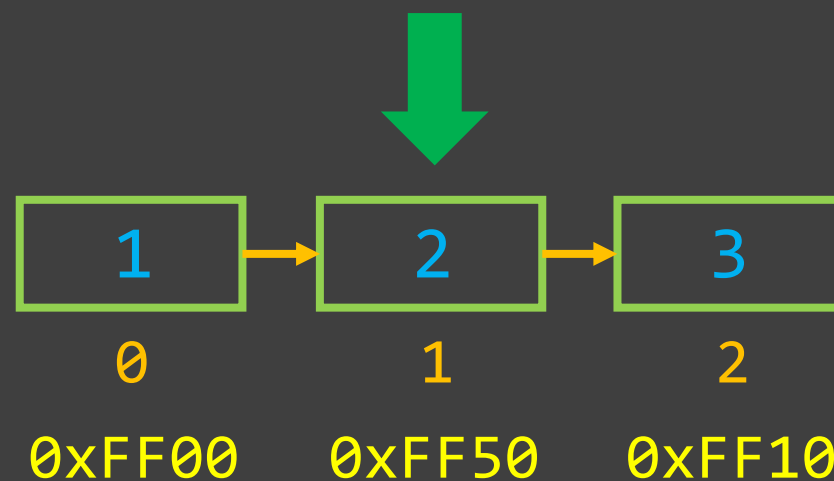
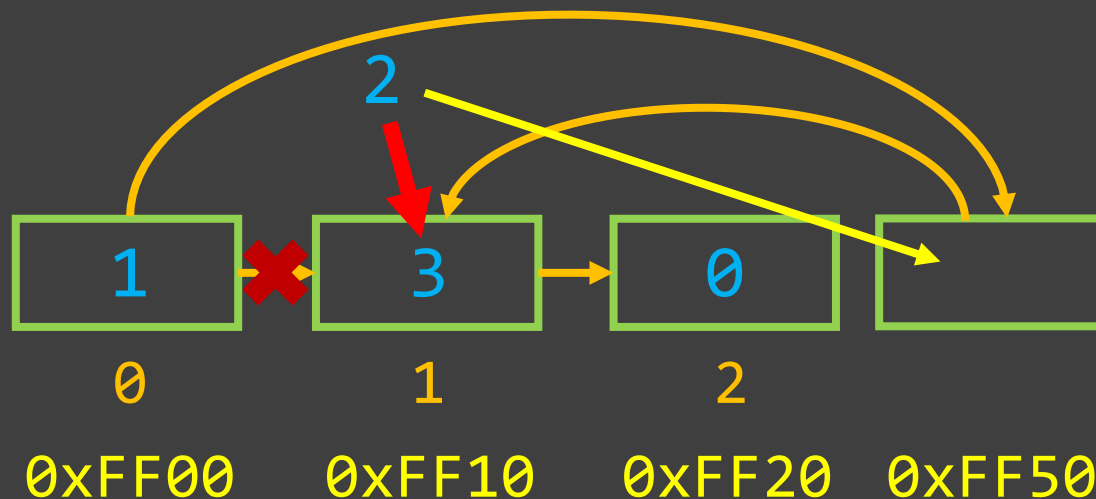
而鏈結串列需要用索引值 0 元素的鏈結尋找索引值 1 的元素  
用索引值 2 元素的鏈結尋找索引值 3 的元素  
持續下去直到找到指定索引值的元素

# 陣列與鏈結串列

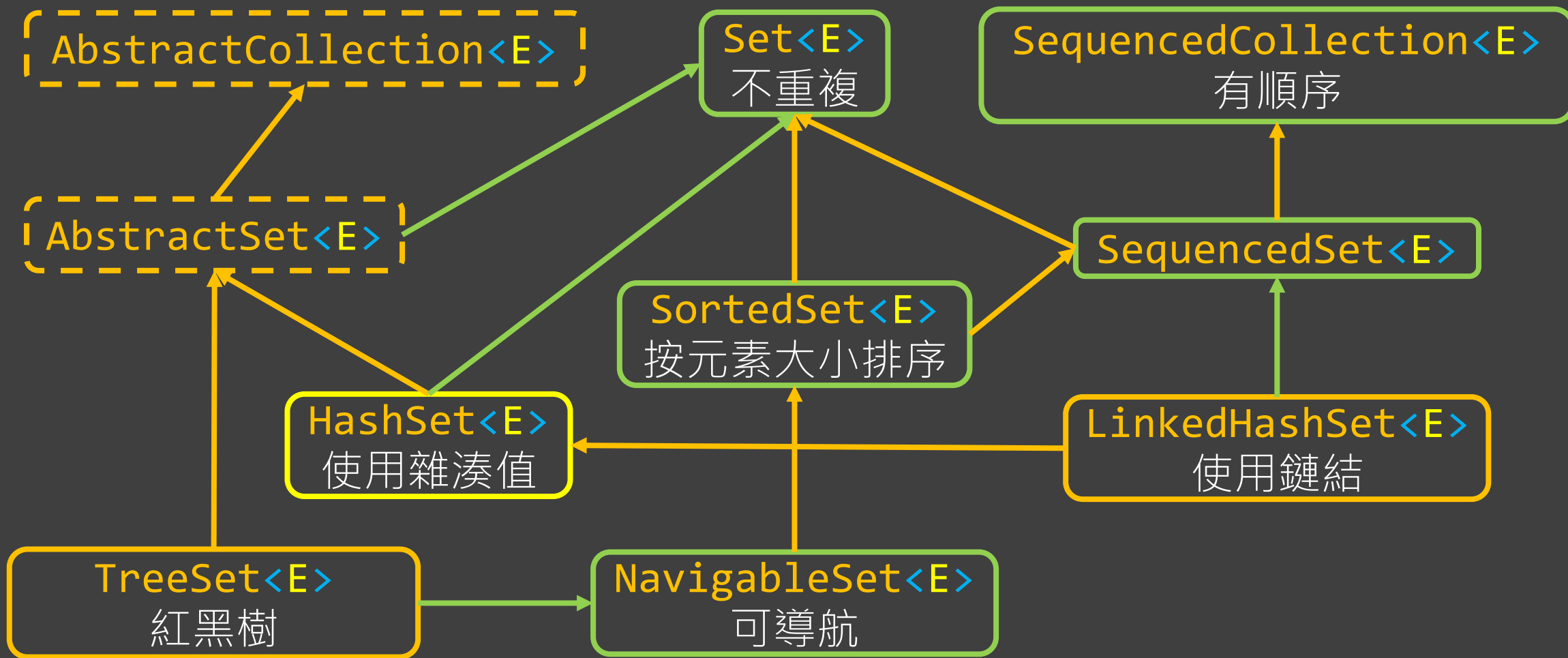
## 陣列



## 鏈結串列



# Set



# HashSet

**Set<E>** 介面的實作類別儲存的元素不可重複，稱為「集合(set)」  
該介面有一個靜態方法 **<E> Set<E> of(E... elements)**

**Set<E>** 介面的實作類別

內部皆是使用對應的 **Map<E, Object>** 來實作

**Set<E>** 的元素作為 **Map<E, Object>** 的鍵

而該鍵對應的值皆參考至同一個 **Object** 物件

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Objects;
import java.util.Set;
```

# HashSet

```
public class Main {
    public static void main(String[] args) {
        HashSet<Person> hashSet = new HashSet<>();
        hashSet.add(new Person(30, "鄭宜"));
        System.out.println(hashSet);
        hashSet.addAll(Set.of(
            new Person(20, "陳生"),
            new Person(30, "鄭宜"),
            new Person(40, "迪鶯"),
            new Person(50, "張棟良")
        ));
        System.out.println(hashSet);
        int ageSum = 0;
        for (Person person : hashSet) {
            ageSum += person.age;
        }
        int ageAvg = ageSum / hashSet.size();
        hashSet.removeIf(person -> person.age >= ageAvg);
        System.out.println(hashSet);
        Iterator<Person> it = hashSet.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
            it.remove();
        }
        System.out.println(hashSet);
    }
}
```

```
[姓名：鄭宜 年齡：30]
[姓名：鄭宜 年齡：30, 姓名：張棟良 年齡：50, 姓名：陳生 年齡：20, 姓名：迪鶯 年齡：40]
[姓名：鄭宜 年齡：30, 姓名：陳生 年齡：20]
姓名：鄭宜 年齡：30
姓名：陳生 年齡：20
[]
```

output

```
class Person {
    protected int age;
    String name;

    Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "姓名：%s 年齡：%d".formatted(name, age);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

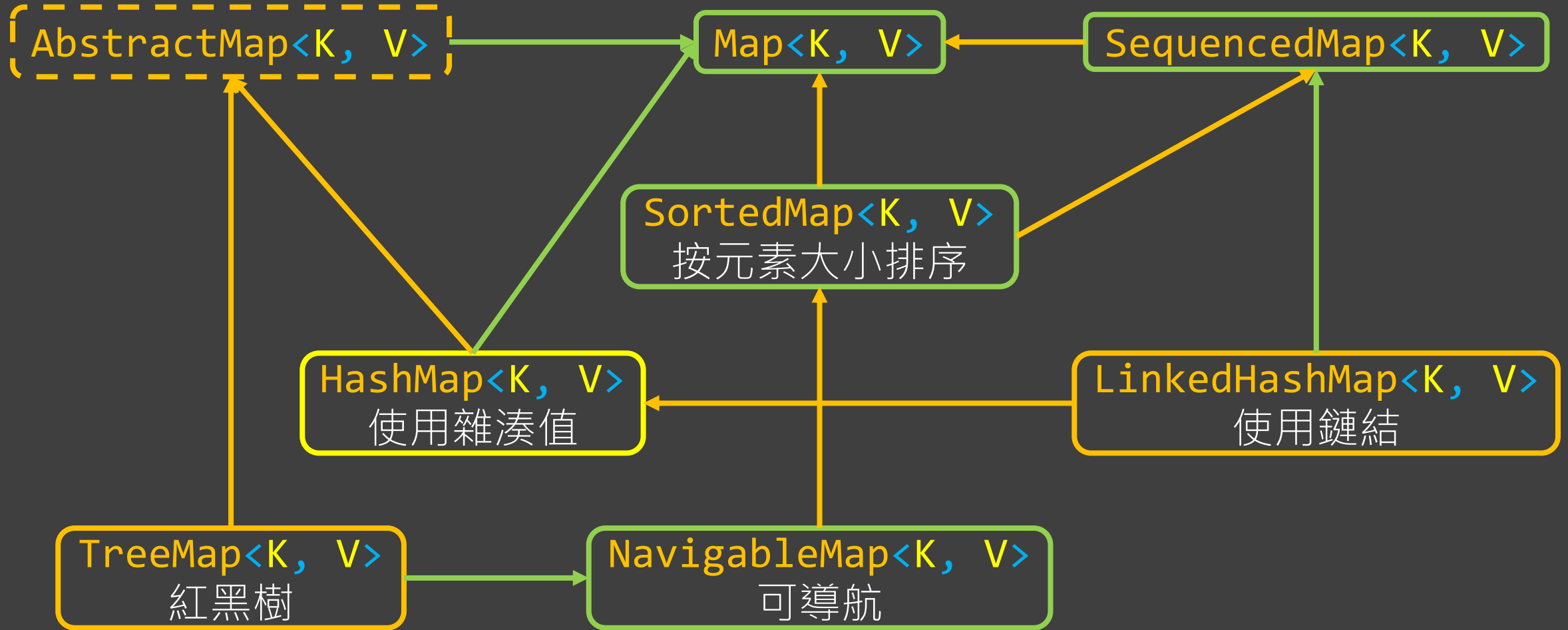
    @Override
    public int hashCode() {
        return Objects.hash(age, name);
    }
}
```



java



# Map



# Map

下方為 `Map<K, V>` 介面部分公開動態方法：

```
int size()、boolean isEmpty()、void clear()  
Collection<V> values()、Set<V> keyset()、Set<Map.Entry<K, V>> entrySet()  
V put(K key, V value)、V remove(Object key)  
void putAll(Map<? extends E, ? extends V> m)  
void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)  
boolean containsKey(Object key)、boolean containsValue(Object value)  
V get(Object key)、V getOrDefault(Object key, V defaultValue)  
void forEach(BiConsumer<? super K, ? super V> action)
```

`Map<K, V>` 的所有實作類別都有一個建構子

其唯一參數為 `Map<? extends K, ? extends V>`

該介面最重要的實作類別就是 `HashMap<K, V>`

# HashMap

**HashMap<K, V>** 使用陣列實作鍵值映射  
並在內部使用雜湊值(**hashCode**)來檢查鍵是否已經存在

雜湊值是經由雜湊函式(**hash function**)計算出的  
具不可逆性，且兩物件相等，雜湊值應相同  
若兩物件不相等時，雜湊值仍然可能出現相同的情況  
稱為雜湊碰撞(**hash collision**)

在 **Java** 中，物件的雜湊函式就是動態方法 **hashCode()**

# HashMap

**HashMap<K, V>** 內部陣列的元素型別為鏈結串列或紅黑樹  
加入新鍵值對時，會先將鍵的雜湊值與內部陣列長度取餘，得到結果 **i**  
然後將鍵值對放入內部陣列索引值 **i** 的鏈結串列或紅黑樹中  
若 **i** 相同的鍵值對數量不超過 **8** 個，則用鏈結串列儲存這些鍵值對  
反之，則使用紅黑樹來儲存這些鍵值對  
在總鍵值對數量達到舊陣列長度的 **75%** 時  
會創建長度為舊陣列長度 **2** 倍的新陣列  
然後將舊陣列內的元素重新計算雜湊值，並加入新陣列

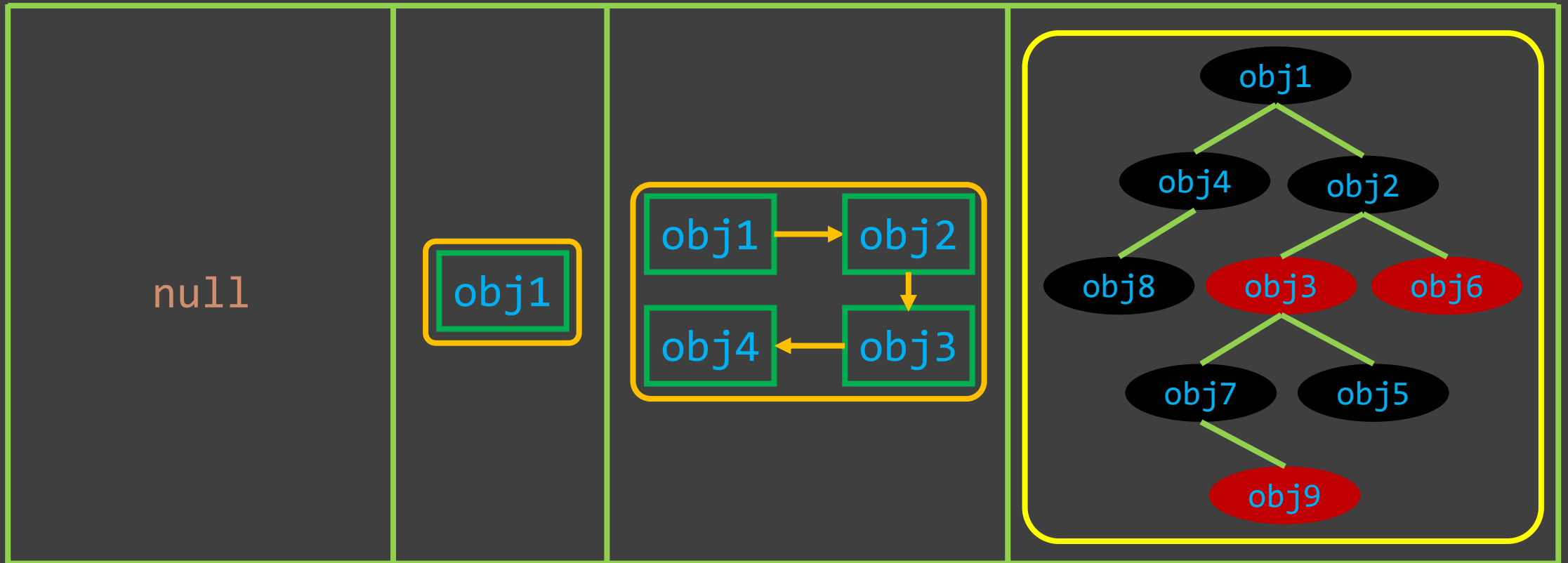
# HashMap

索引值0 ~ 28

索引值29

索引值30

索引值31



# HashMap

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Objects;

public class Main {
    public static void main(String[] args) {
        HashMap<Integer, Person> hashMap = new HashMap<>();
        hashMap.put(0, new Person(30, "張雨"));
        hashMap.put(1, new Person(35, "吳棕憲"));
        hashMap.put(2, new Person(40, "劉德涓"));
        System.out.println(hashMap);
        for (Iterator<Map.Entry<Integer, Person>> it =
            hashMap.entrySet().iterator(); it.hasNext(); ) {
            Map.Entry<Integer, Person> personEntry = it.next();
            personEntry.getValue().age += 10;
        }
        hashMap.forEach((id, person) ->
            System.out.println("編號：" + id + " " + person));
    }
}
```

```
{0=姓名：張雨 年齡：30, 1=姓名：吳棕憲 年齡：35, 2=姓名：劉德涓 年齡：40}
編號：0 姓名：張雨 年齡：40
編號：1 姓名：吳棕憲 年齡：45
編號：2 姓名：劉德涓 年齡：50
```

output

```
class Person {
    int age;
    String name;

    Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "姓名：%s 年齡：%d".formatted(name, age);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(age, name);
    }
}
```



java

# 集合框架工具類別

集合框架的工具類別是 `java.util.Collections`，其中定義了許多關於集合框架的公開靜態方法，如：

```
<T> Comparator<T> reverseOrder()、<T> Comparator<T> reverseOrder(Comparator<T> cmp)
<T> boolean addAll(Collection<? super T> c, T... elements)
<T> void fill(List<? super T> list, T obj)、<T> List<T> nCopies(int n, T o)
<T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
<T> void copy(List<? super T> dest, List<? extends T> src)
<T> List<T> emptyList()、<T> Set<T> emptySet()、<K, V> Map<K, V> emptyMap()
<T extends Comparable<? super T>> T max(Collection<? extends T> coll)
<T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
<T extends Comparable<? super T>> T min(Collection<? extends T> coll)
<T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
void shuffle(List<?> list)、<T> void sort(List<T> list, Comparator<? super T> c)
<T extends Comparable<? super T>> void sort(List<T> list)
```

# 集合框架工具類別

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Objects;

public class Main {
    public static void main(String[] args) {
        ArrayList<Person> people = new ArrayList<>();
        Collections.addAll(people,
            new Person(50, "善依純"),
            new Person(10, "蕭亞瑄"),
            new Person(40, "郭靖"),
            new Person(30, "陳生"),
            new Person(20, "徐懷豫"));
        System.out.println(Collections.min(people) +
            ", " + Collections.max(people));
        Comparator<Person> reversedComp =
            Collections.reverseOrder(Person::compareTo);
        Collections.sort(people, reversedComp);
        System.out.println(people);
        System.out.println(Collections.binarySearch(people,
            new Person(20, "徐懷豫"), reversedComp));
        Collections.shuffle(people);
        System.out.println(people);
    }
}
```

```
姓名：蕭亞瑄 年齡：10, 姓名：善依純 年齡：50
[姓名：善依純 年齡：50, 姓名：郭靖 年齡：40, 姓名：陳生 年齡：30, 姓名：徐懷豫 年齡：20, 姓名：蕭亞瑄 年齡：10]
3
[姓名：善依純 年齡：50, 姓名：郭靖 年齡：40, 姓名：蕭亞瑄 年齡：10, 姓名：陳生 年齡：30, 姓名：徐懷豫 年齡：20]
```



```
class Person implements Comparable<Person> {
    int age;
    String name;

    Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "姓名：%s 年齡：%d".formatted(name, age);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(age, name);
    }

    @Override
    public int compareTo(Person o) {
        return age - o.age;
    }
}
```

java

output



# Stream

`java.util.stream.Stream<T>` 是用來對資料進行一連串處理的介面

下方為部分公開動態方法：`long count()`

篩選：

`Stream<T> distinct()`

`Stream<T> filter(Predicate<? super T> predicate)`

映射：

`<R> Stream<R> map(Function<? super T, ? extends R> mapper)`

`DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)`

`IntStream mapToInt(ToIntFunction<? super T> mapper)`

`LongStream mapToLong(ToLongFunction<? super T> mapper)`

# Stream

排序：

```
Stream<T> sorted()
```

```
Stream<T> sorted(Comparator<? super T> comparator)
```

查找：

```
boolean allMatch(Predicate<? super T> predicate)
```

```
boolean anyMatch(Predicate<? super T> predicate)
```

```
boolean noneMatch(Predicate<? super T> predicate)
```

```
Optional<T> max(Comparator<? super T> comparator)
```

```
Optional<T> min(Comparator<? super T> comparator)
```

```
Optional<T> findFirst()
```

# Stream

走訪：

```
Stream<T> peek(Consumer<? super T> action)
```

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

```
void forEach(Consumer<? super T> action)
```

蒐集：

```
<R> R collect(Supplier<R> supplier,  
              BiConsumer<R, ? super T> accumulator,  
              BiConsumer<R, R> combiner)
```

```
<R, A> R collect(Collector<? super T, A, R> collector)
```

```
List<T> toList()、Object[] toArray()
```

# Stream

**Stream<T>** 方法的呼叫

最後一個呼叫一定要是不返回 **Stream<T>** 的方法(終端方法)

否則前面呼叫的方法(中介操作)皆不會實際執行

許多資料類別都支援使用 **Stream<T>**，如：陣列、集合框架等

呼叫各類別的 **Stream<E> stream()** 方法

就可以取得 **Stream<T>** 介面的實作類別(來源)並進行管線操作

(陣列使用 **<T> Stream<T> Arrays.stream(T[] array)** 取得)

# Stream

```
import java.util.ArrayList;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> arrayList = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) arrayList.add(scanner.next());
        System.out.println(arrayList.stream().reduce(String::concat).orElse(""));
        System.out.println(arrayList.stream()
            .map((str) -> Character.toUpperCase(str.charAt(0)))
            .collect(StringBuilder::new, StringBuilder::append, StringBuilder::append)
            .toString());
    }
}
```

as SOON as POSSIBLE  
^D  
asSOONasPOSSABLE  
ASAP

console



java