

# PD2 Lecture OOP - Representation of OO Design and Powerful Libraries/Functionalities



"Don't Reinvent the Wheels", Generated by DALL.E

## Revisit - static and final in the class concept

在Java中，`static` 和 `final` 都是重要的關鍵字。

`static` 關鍵字用於表示一個成員變數或者方法屬於類別本身，而不是類別的instance-level。這意味著，不論我們創建多少個類別的object，靜態變數都只有一份資料，所

有objectobject共享該靜態變數。靜態method也是類別方法，它不依賴於任何instance，所以靜態method中不能訪問類別中的非靜態變數。例如，我們有一個名為MyClass的類別，該類別中有一個靜態方法staticMethod和一個非靜態變數nonStaticVariable。

```
public class MyClass {  
    private int nonStaticVariable;  
  
    public static void staticMethod() {  
        System.out.println(nonStaticVariable); // 這行會導致  
        編譯錯誤  
    }  
}
```

在上述程式中，staticMethod試圖access nonStaticVariable。但是，這會導致編譯錯誤，因為靜態方法不能訪問非靜態變數。這是因為靜態方法屬於類別本身（class-level），而nonStaticVariable變數屬於類別的實例(instance-level)。因此，當staticMethod被調用時，nonStaticVariable可能還未被初始化。為了避免這種情況，Java禁止靜態方法訪問非靜態變數。



當staticMethod被調用時，nonStaticVariable可能還未被初始化？為什麼說"可能"呢？有什麼情況會已被初始化了呢？

**生命週期：**靜態變數和靜態方法的生命週期從類別被JVM loading開始，到被JVM卸載結束。



簡單的想，你也可以將static變數，思考成是所有這個class的物件，都共享這個member data。

final關鍵字用於表示一個變數是read-only的，或者一個method或class是不能被override的。如果一個變數被final修飾，那麼它只能被賦值一次，賦值後其值就不能再被改變。如果一個方法被final修飾，那麼它不能被子類別重寫（能被子類別繼承）。如果一個類別被final修飾，那麼它不能被繼承。

**生命週期：**在Java中，被final關鍵字修飾的變數或object的生命週期從它被初始化開始，到object被GC收回結束。

## Example of static members

```
class Test {
    public int x; // 個別物件擁有一份
    public static int y; // 所有此類別物件共享

    // 具有參數的建構方法
    public Test(int x,int y) {
        this.x = x;
        this.y = y;
    }
    public static void staticMethod() {
        System.out.println("This is a static method!");
    }

    // 轉成字串
    public String toString() {
        return "(x,y):(" + x + "," + y + ")";
    }
}

public class StaticMember {

    public static void main(String[] argv){
        //你可以沒有create任何的object就存取static member
        System.out.println(Test.y); //初始為0
        Test.staticMethod(); //可以以class-level方式呼叫static method
        Test a = new Test(100,40);
        Test b = new Test(200,50);
        Test c = new Test(300,60);
        a.staticMethod(); //也可以以object-level方式呼叫static method
        System.out.println("物件a" + a);
        System.out.println("物件b" + b);
        System.out.println("物件c" + c);
    }
}
```

## Static\_INITIALIZER

在Java中，我們可以使用靜態初始化區塊（Static Initializer Block）來初始化靜態變數。這個區塊只會在class被載入到JVM時執行一次。

以下是一個靜態初始化區塊的範例：

```
public class MyClass {
    static int[] array;

    static {
        array = new int[5];
        for (int i = 0; i < array.length; i++) {
            array[i] = i * 10;
        }
    }

    public static void main(String[] args) {
        for (int i : array) {
            System.out.println(i);
        }
    }
}
```

在以上的範例中，我們首先宣告了一個靜態整數陣列 `array`。然後，我們使用靜態初始化區塊來初始化這個陣列。在靜態初始化區塊中，我們創建了一個長度為5的新陣列並將其賦值給 `array`，然後將每個元素的值設定為其索引值的10倍。最後，在 `main` 方法中，我們使用for-each循環來輸出陣列中的每個元素。

## Introduction to UML

### UML (Unified Modeling Language)

UML 是一種用於軟體系統設計的視覺化語言，它可用於描述系統的結構、行為和交互作用。

在 Java 中，可以使用 UML 圖表來描述類別、物件、關係和行為。

統一建模語言（UML）是一種用於軟體工程的標準化建模語言，它提供了一套圖形化表示用於系統的分析、設計、實現和文檔化。UML中的一些主要圖表包括類圖

（Class Diagram）、關係圖（通常是類別圖的一部分，展示類別之間的關係）、序列圖（Sequence Diagram）等。

以下是以車子的元件為例，介紹這三種UML圖及對應的Java範例和Mermaid圖。

## 1. 類別圖 (Class Diagram)

類圖是用來描述系統中的類以及它們之間的關係。在車子的例子中，我們可以有 `Car`、`Engine`、`Wheel` 等類。

### Java範例

```
class Car {
    Engine engine;
    Wheel[] wheels = new Wheel[4];
}

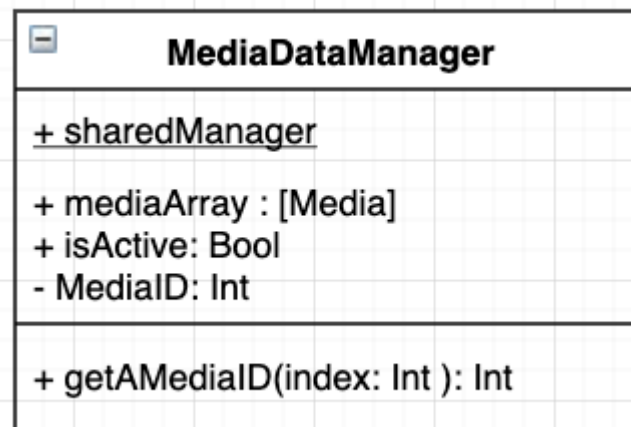
class Engine {
    int horsepower;
}

class Wheel {
    int size;
}
```

### 對應的Mermaid圖

```
classDiagram
    class Car {
        -Engine engine
        -Wheel[] wheels
    }
    class Engine {
        -int horsepower
    }
    class Wheel {
        -int size
    }
    Car --* Engine : has
    Car --* Wheel : has 4
```

## 類別描述



### 類別範例

三個區域由上而下分別代表

1. Name — 必填的名稱
2. Attributes — 屬性，冒號後表示型別
3. Methods — 方法，冒號後表示回傳值

其中屬性與方法可以在前面加上前綴符號，表示其scope

## 2. 關係圖 (Relationship Diagram)

關係圖通常是類圖的一部分，用於展示類別之間的不同類型的關係，如關聯 (association)、聚合 (aggregation)、組合 (composition) 和繼承 (inheritance)。在車子例子中，`Car` 與 `Engine` 之間是組合關係，`Car` 與 `wheel` 也是組合關係。這部分已經在上面的類別圖中展示。

在Java中，關聯 (Association) 關係可以表示為兩個類別中的一個包含另一個類別的引用。下面是一個範例代碼，其中 `Student` 類與 `Teacher` 類之間存在關聯關係：

```
public class Teacher {
    private String name;

    public Teacher(String name) {
        this.name = name;
    }

    public String getName() {
```

```

        return this.name;
    }
}

public class Student {
    private String name;
    private Teacher teacher; // 這裡就是關聯關係

    public Student(String name, Teacher teacher) {
        this.name = name;
        this.teacher = teacher;
    }

    public String getTeacherName() {
        return this.teacher.getName();
    }
}

```

在這個例子中，每個 `Student` 都有一個 `Teacher` 的引用，這就構成了一種關聯關係。

```

erDiagram
    STUDENT ||--|{ TEACHER : has

```

### 3. 序列圖 (Sequence Diagram)

序列圖是用來描述object之間在時間序列上的交互關係。這裡描述當一個 `Driver` 啟動 `Car` 的過程。

#### Java範例

```

class Driver {
    void drive(Car car) {
        car.start();
    }
}

class Car {
    Engine engine;
    void start() {

```



```

        engine.start();
    }
}

class Engine {
    void start() {
        // 啟動引擎
    }
}

```

## 對應的Mermaid圖

```

sequenceDiagram
    participant Driver
    participant Car
    participant Engine
    Driver->>Car: start()
    Car->>Engine: start()
    Note right of Engine: 引擎啟動


```

這些範例展示了UML圖和對應的Java程式之間的關聯，以及如何使用Mermaid語法來視覺化這些概念。這些工具和技術有助於在軟體開發過程中提供清晰的設計和溝通機制。

## PlanUML for VS Code


[TIL] 在 vscode 上面安裝並且使用 PlantUML

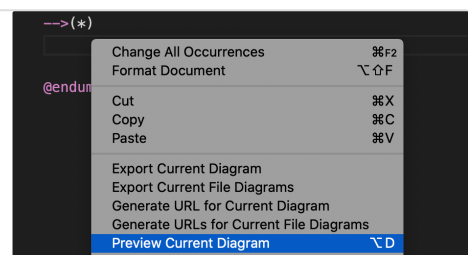
寫作技術文章的時候，經常需要各種 UML (Unified Modeling Language) 所繪製出來的圖形。雖然學生時代都會學過這個，但是工作之後其實不容易透過良好的工具來繪製。

 [https://www.evanlin.com/til-vscode-plantuml/?source=post\\_page-----48b7ca1922cd-----](https://www.evanlin.com/til-vscode-plantuml/?source=post_page-----48b7ca1922cd-----)

planUML : 透過 VScode畫 UML 圖

VSCode 套件 安裝 plantuml

 <https://medium.com/kidd88/planuml-透過-vscode畫-uml-圖-48b7ca1922cd>





## Example of Car

To demonstrate the relationships between objects, we can use a simple example of car parts. This example will include several basic classes: Car, Engine, Wheel, and Transmission. In this model, a car contains one engine, one transmission system, and several wheels. This illustrates composition and aggregation relationships.

```
class Car {
    private Engine engine;
    private Transmission transmission;
    private Wheel[] wheels;

    public Car() {
        engine = new Engine();
        transmission = new Transmission();
        wheels = new Wheel[4]; // 假設每輛車有四個輪胎
        for (int i = 0; i < wheels.length; i++) {
            wheels[i] = new Wheel();
        }
    }
    // 其他方法
}
```

//These methods provide a simulation of basic operations for each component, such as starting and stopping the engine, changing the gears of the transmission system, and inflating the tires. Through such practices, students can gain a deeper understanding of the concept of classes in object-oriented programming and how to implement methods within classes to perform operations.

```
class Engine {
    private boolean isRunning;

    public Engine() {
        this.isRunning = false;
    }
}
```

```

    public void start() {
        isRunning = true;
        System.out.println("Engine started.");
    }

    public void stop() {
        isRunning = false;
        System.out.println("Engine stopped.");
    }

    public boolean isRunning() {
        return isRunning;
    }
}

class Transmission {
    private int gear;

    public Transmission() {
        this.gear = 0; // Assuming 0 is neutral
    }

    public void changeGear(int gear) {
        this.gear = gear;
        System.out.println("Shifted to gear " + gear +
".");
    }

    public int getGear() {
        return gear;
    }
}

class Wheel {
    private double pressure;

    public Wheel() {
        this.pressure = 32.0; // Assuming initial tire pres

```

```

sure is 32psi
    }

    public void inflate(double psi) {
        pressure += psi;
        System.out.println("Tire inflated by: " + psi + "psi. Current pressure: " + pressure + "psi.");
    }

    public double getPressure() {
        return pressure;
    }
}

```

## UML Class Diagram

```

@startuml
class Car {
    -Engine engine
    -Transmission transmission
    -Wheel[] wheels
}

class Engine {
    -boolean isRunning
    +void start()
    +void stop()
    +boolean isRunning()
}

class Transmission {
    -int gear
    +void changeGear(int)
    +int getGear()
}

class Wheel {
    -double pressure
}

```

```

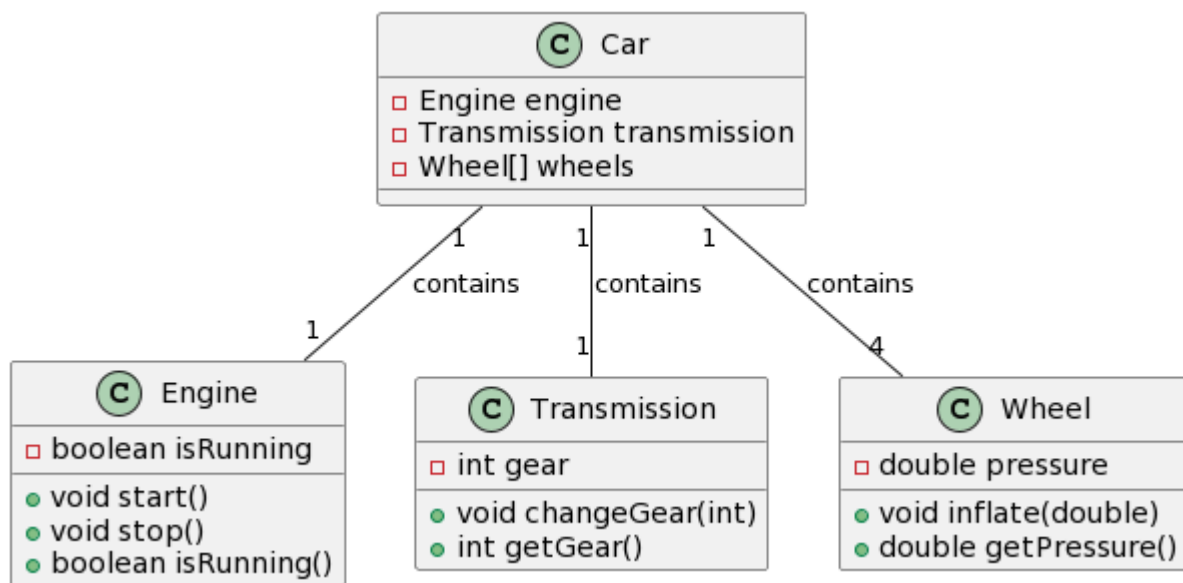
+void inflate(double)
+double getPressure()
}

```

```

Car "1" -- "1" Engine : contains
Car "1" -- "1" Transmission : contains
Car "1" -- "4" Wheel : contains
@enduml

```



Try <https://www.planttext.com/>

## Revisit Java Foundation for Class

### Object

在Java中所有的物件，其base object都是 `java.lang.Object` (所有的class都繼承自 `java.lang.Object`)。

方法名稱	描述
<code>public boolean equals(Object obj)</code>	指示某個其他物件是否與此物件“相等”。
<code>public int hashCode()</code>	返回該物件的hash值。
<code>public String toString()</code>	返回該物件的string表示。
<code>protected Object clone() throws</code>	創建並返回此物件的 <b>一個新的實體</b>

方法名稱	描述
CloneNotSupportedException	<code>instance</code> 。
protected void finalize() throws Throwable	當GC確定該物件沒有更多的reference時，物件被垃圾回收時會call此方法。

## equals()

在Java中，如果想要比較兩個物件是否完全相等，我們需要覆寫或實作 `equals()` 方法，而非使用 `==` 運算符號。



`==` 運算符號比較的是兩個物件的reference位置，也就是它們在記憶體中的位置，而不是它們的內容。因此，即使兩個物件的內容完全相同，如果它們在記憶體中的位置不同，`==` 也會返回 `false`。

反之，`equals()` 方法用於比較兩個物件的內容是否相等。



但如果我們想要比較的物件類別沒有override `equals()`，那麼它將會使用上層的class的 `equals()` 如 `java.lang.Object` 類別的 `equals()`，而如果是 `java.lang.Object` 類別的 `equals()`，則這個method的行為與 `==` 相同。

因此，我們需要在我們的物件類別中覆寫 `equals()` 來確保它比較的是物件的內容。

例如：

```
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
```

```

        MyClass myClass = (MyClass) obj;
        return value == myClass.value;
    }
}

```

在上述的例子中，`MyClass` 覆寫了 `equals()` 方法以比較物件的 `value` 屬性。因此，如果兩個 `MyClass` 物件的 `value` 相等，那麼 `myClass1.equals(myClass2)` 將返回 `true`，即使 `myClass1` 和 `myClass2` 在記憶體中的位置不同。

## getClass()

在 Java 中，`getClass()` 是 `java.lang.Object` 類別的一個方法，所有的物件都繼承自 `java.lang.Object` 類別，因此都可以使用 `getClass()` 方法。這個方法用來取得該物件的 `Class` 物件，該 `Class` 物件代表了該物件的實際類別（class）。

`getClass()` 方法的一個常見的使用情境是在動態類別識別(dynamic class identification and reflection)。例如，當我們需要判斷一個物件是否屬於某個類別，或者當我們想要動態地取得一個物件的方法和屬性時，我們可以使用 `getClass()` 方法。

以下是一個使用 `getClass()` 的範例：

```

public class MyClass {
    public static void main(String[] args) {
        String str = "Hello, world!";
        Class<?> strClass = str.getClass();
        System.out.println("The class of str is " + strClass.getName());
    }
}

```

在這個例子中，我們首先創建了一個 `String` 物件 `str`，然後使用 `getClass()` 方法取得了 `str` 的 `Class` 物件，並將其儲存在 `strClass` 變數中。最後，我們輸出了 `strClass` 的名稱，這將輸出 `"java.lang.String"`，這正是 `str` 物件的類別名稱。

其中 `Class<?> strClass = str.getClass();` 是在獲取 `str` object 的類別。這裡的 `getClass()` 是一個method，其主要功能是獲取object的類別。這個方法返回一個 `Class` object，該 `str` object 包含了調用object的類別的各種信息，如類別名稱、方法、屬性等。

在這裡，`?` 是一個泛型符號(generic symbols)，代表的是未知類型。當我們不確定或者不需要關心具體的類型時，就可以使用 `?` 來表示。在這裡，`Class<?>` 表示的是一

個未知類型的 `Class` object。因為 `getClass()` 方法返回的類型是當前object的類別，而這個類別在compile時可能是未知的，所以我們使用 `Class<?>` 來接收返回值。

## clone()

在Java中，`Object` 類別提供了一個 `clone()` 方法，可以用來創建並返回該物件的一個副本。這個方法預設提供的是shallow copy，稱之為淺層拷貝，也就是說，如果物件的欄位是基本型態primitive type，那麼會直接複製它們的值；如果物件的欄位是reference型態，那麼只會複製這個reference，而不會複製參考指向的物件。

如果我們希望實現deep copy（深層拷貝），也就是說，不僅複製物件本身，還複製物件所參考的物件，那麼我們就需要覆寫 `clone()` 方法。

以下是一個範例，展示如何覆寫 `clone()` 方法以實現深層拷貝：

```
public class MyClass implements Cloneable {
    private int[] array;

    public MyClass(int[] array) {
        this.array = array;
    }

    // Getter for array
    public int[] getArray() {
        return array;
    }

    @Override
    public MyClass clone() {
        int[] clonedArray = array.clone(); // This line creates a copy of the array. It is a primitive-type array and clone() can return an array like deep-copy
        return new MyClass(clonedArray); // A new MyClass object is created with the cloned array.
    }
}
```

在上述程式碼中，`MyClass` 類別覆寫了 `clone()` 方法，並在該方法中首先複製了 `array`，然後用這個複製的 `array` 創建了一個新的 `MyClass` 物件。由於 `array` 被複製了，所以這個 `clone()` 方法實現了深層拷貝。





`Cloneable` 介面在Java中是一種標記性介面(marker interface)，它不包含任何方法。當一個類別實作了 `Cloneable` 介面後，即表示該類別具有被複製的能力，並且可以通過該類別的`clone()`方法來創建物件的複製。然而，如果一個類別沒有實現 `Cloneable` 介面，則在呼叫該類別的`clone()`方法時將拋出 `CloneNotSupportedException`。

在Java中，雖然每個物件都可以呼叫 `clone()` 方法，但該方法在 `Object` 類別中是 `protected` 的。而如果沒有明確地在一個類別中override `clone()` 方法，則不能夠在該類別的instance上呼叫 `clone()` 方法。另一方面，`Cloneable` 介面是一種標記介面，用於指示一個類別的instance可以被安全地複製。如果一個類別實現了 `Cloneable` 介面但並未覆寫 `clone()` 方法，當呼叫 `clone()` 方法時將呼叫 `Object` 類別的 `clone()` 方法，並且不會拋出 `CloneNotSupportedException`。然而，如果一個類別並未實現 `Cloneable` 介面，則當呼叫 `clone()` 方法時將拋出 `CloneNotSupportedException`。因此，`Cloneable` 介面的存在是為了明確地表明一個類別的instance可以被複製。



在Java中，如果一個method被定義為private，請問它還可以被子類別override嗎？那如果在子類別中定義了一樣的method name (argument也一樣)，會怎麼樣呢？



深層拷貝 (Deep Copy) 與淺層拷貝 (Shallow Copy) 是程式語言中對物件拷貝方式的一種區分。深層拷貝是指在記憶體中創建一個新的物件，並將原物件中的非靜態欄位一一拷貝到新物件中。如果原物件的欄位是值型態，就直接拷貝值；如果原物件的欄位是reference型態，那麼就創建一個新的物件，並將這個新物件的reference賦值給新物件的相對應欄位。



在Java中，對於 `int[]` 陣列的 `clone()` 操作，會進行deep copy (深層拷貝)，因為它是一個基本型態的陣列，所以 `clone()` 會創建一個新的陣列並複製所有的元素。

然而，對於 `Object[]` 陣列的 `clone()` 操作，則是shallow copy (淺層拷貝)。這是因為 `Object[]` 陣列的元素是reference型態，`clone()` 僅僅會複製這些reference，而不會複製reference所指向的物件。因此，原陣列和複製後的陣列中的元素會指向同一個物件。

```

public class DeepCopy {
    int[] data;

    // Makes a deep copy of values
    public DeepCopy(int[] values) {
        data = new int[values.length];
        for (int i = 0; i < data.length; i++) {
            data[i] = values[i];
        }
    }

    public void showData() {
        for (int i = 0; i < data.length; i++) {
            System.out.println(data[i]);
        }
    }
}

```

淺層拷貝則是在記憶體中創建一個新的物件，並將原物件中的非靜態欄位一一拷貝到新物件中。如果原物件的欄位是primitive型態，就直接拷貝值；如果原物件的欄位是reference型態，那麼就將這個欄位的參考賦值給新物件的相對應欄位。簡而言之，淺層拷貝只拷貝reference，並不創建新的物件。

```

public class ShallowCopy {
    int[] data;

    // Makes a shallow copy of values
    public ShallowCopy(int[] values) {
        data = values;
    }

    public void showData() {
        for (int i = 0; i < data.length; i++) {
            System.out.println(data[i]);
        }
    }
}

```

`System.arraycopy` 是Java中提供的拷貝陣列的方法，它提供的是淺層拷貝。也就是說，對於reference型態的欄位，該方法只會拷貝reference，而不會創建新的物件。

```
int[] original = new int[] {1, 2, 3, 4, 5};
int[] copied = new int[5];
System.arraycopy(original, 0, copied, 0, original.length);
```

在這個例子中，`System.arraycopy` 方法將 `original` 陣列中的元素拷貝到 `copied` 陣列中。但這只是值的拷貝，如果 `original` 陣列中的元素是reference型態，則 `copied` 陣列中的相對應欄位將與 `original` 陣列中的欄位指向同一個物件。

## 二維陣列的Shallow Copy

```
public class ShallowCopyExample {
    public static void main(String[] args) {
        // 建立原始二維陣列
        int[][] original = {{1, 2, 3}, {4, 5, 6}};

        // 創建原始陣列的淺拷貝
        int[][] copied = original.clone();

        // 輸出原始陣列和複製陣列的第一個元素
        System.out.println("Before modification:");
        System.out.println("original[0][0] = " + original
[0][0]);
        System.out.println("copied[0][0] = " + copied[0]
[0]);

        // 修改複製陣列的第一個元素
        copied[0][0] = 100;

        // 再次輸出原始陣列和複製陣列的第一個元素
        System.out.println("After modification:");
        System.out.println("original[0][0] = " + original
[0][0]);
        System.out.println("copied[0][0] = " + copied[0]
[0]);
```

```
}  
}
```

當你執行這個程式時，將會看到以下的輸出：

```
Before modification:  
original[0][0] = 1  
copied[0][0] = 1  
After modification:  
original[0][0] = 100  
copied[0][0] = 100
```

這個輸出證明了在修改複製陣列的元素之後，原始陣列對應的元素也被修改了。這就是淺層拷貝（Shallow Copy）的特性。



`array.clone()` 和 `System.arraycopy()` 都可用於複製陣列。然而，主要的差異在於 `array.clone()` 創建了一個新的陣列，並返回這個陣列的reference，而 `System.arraycopy()` 則需要一個已經存在的陣列才能將數據複製到其中。

如果我們想要實現二維陣列的深層拷貝，我們需要為每個一維陣列創建一個新的陣列，並將原始二維陣列中的元素拷貝到新的二維陣列中。以下是一個實現二維陣列深層拷貝的Java程式：

```
public class DeepCopyExample {  
    public static void main(String[] args) {  
        // 建立原始二維陣列  
        int[][] original = {{1, 2, 3}, {4, 5, 6}};  
  
        // 創建原始陣列的深層拷貝  
        int[][] copied = new int[original.length][];  
        for (int i = 0; i < original.length; i++) {  
            copied[i] = original[i].clone();  
        }  
  
        // 輸出原始陣列和複製陣列的第一個元素  
        System.out.println("Before modification:");  
        System.out.println("original[0][0] = " + original
```

```

[0][0]);
    System.out.println("copied[0][0] = " + copied[0]
[0]);

    // 修改複製陣列的第一個元素
    copied[0][0] = 100;

    // 再次輸出原始陣列和複製陣列的第一個元素
    System.out.println("After modification:");
    System.out.println("original[0][0] = " + original
[0][0]);
    System.out.println("copied[0][0] = " + copied[0]
[0]);
    }
}

```

當你執行這個程式時，將會看到以下的輸出：

```

Before modification:
original[0][0] = 1
copied[0][0] = 1
After modification:
original[0][0] = 1
copied[0][0] = 100

```

這個輸出證明了在修改複製陣列的元素之後，原始陣列對應的元素並沒有被修改。這就是深層拷貝（Deep Copy）的特性。

## hashCode()

在Java中，`hashCode()` 方法用於返回object的hash code值。hash code值是一個整數，通常由object的memory address計算得出。這個方法主要用於在hash表中查找object，例如，當你在一個 `HashMap` 中插入或查找一個key-value pair時，就會用到這個方法。

在Java中，如果兩個object根據 `equals()` 方法是相等的，那麼這兩個object的 `hashCode()` 方法必須返回相同的結果。然而，反過來並不一定成立，也就是說，如果兩個object的 `hashCode()` 方法返回相同的結果，並不意味著這兩個object根據 `equals()` 方法是相等的。因此，當覆寫 `equals()` 方法時，我們通常也需要覆寫 `hashCode()` 方法，以保證相等的object返回相同的hash code值。



如果你重寫了 `equals()` 方法，但沒有同時改掉 `hashCode()`，可能會導致不一致的行為，特別是在使用像是 `HashMap` 或 `HashSet` 等Java集合類別時。這是因為這些類別依賴於 `hashCode()` 方法來決定物件應該存放在集合的哪一部分。如果相等的物件（根據 `equals()` 方法）沒有相同的 hash code，那麼這些集合可能無法正確地找到或儲存物件，會導致預期之外的行為。

以下是一個範例，展示了如何在自定義的類別中覆寫 `hashCode()` 方法：

```
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        MyClass myClass = (MyClass) obj;
        return value == myClass.value;
    }

    @Override
    public int hashCode() {
        return value;
    }
}
```

在上述程式碼中，`MyClass` 類別override了 `hashCode()` 方法，並將hash code值設定為 `value` 的值。因此，對於 `value` 相等的 `MyClass` object，它們的 `hashCode()` 方法將返回相同的結果。

## String

## How to create string

方法	描述	例子
直接指定	這是最簡單的方式，只需要直接將字串指定給一個String變數。	<pre>String str = "Hello";</pre>
使用 new	使用 new 關鍵字創建一個新的 String 物件。	<pre>String str = new String("Hello");</pre>
字元陣列	使用字元陣列創建一個String物件。	<pre>char[] charArray = {'H', 'e', 'l', 'l', 'o'}; String str = new String(charArray);</pre>
字串串接	可以使用 + 符號串接字串。	<pre>String str = "Hello" + "World";</pre>
使用StringBuilder	使用StringBuilder類別可以動態建立和修改字串。	<pre>StringBuilder sb = new StringBuilder(); sb.append("Hello"); String str = sb.toString();</pre>
使用StringBuffer	使用StringBuffer類別也可以動態建立和修改字串，並且是執行緒安全的。	<pre>StringBuffer sb = new StringBuffer(); sb.append("Hello"); String str = sb.toString();</pre>



在 Java 中，"+" 操作operator可以用來連接兩個字串，這是 Java 特別為 String 類型提供的一個特例。儘管 Java 並不支持操作operator重載（operator overloading），但對於 String 類型的 "+" 操作，Java 做了特殊的處理。當 "+" 操作的其中一個unit是 String 類型時，Java 會將另一個unit轉換為 String 類型，然後將兩個 String 進行連接。這就是為什麼我們可以在 Java 中使用 "+" 來連接兩個字串的原因。

## String Reference Concept

Recall：當我們使用 `==` 運算子來比較兩個String object時，我們實際上是在比較兩個object的reference，比較其指向的記憶體地址，而不是指它們的內容。以下是一個相關的範例：

```
String str1 = new String("Hello");
String str2 = new String("Hello");

if (str1 == str2) {
```



```
        System.out.println("兩個字串相同");
    } else {
        System.out.println("兩個字串不同");
    }
}
```

在這個例子中，雖然 `str1` 和 `str2` 的內容是相同的，但是它們實際上是兩個不同的物件，所以當我們使用 `==` 運算子來比較它們時，結果會顯示 "兩個字串不同"。

## ? Recall cases of String Pool

如果我們想要比較兩個String object的內容，我們應該使用 `.equals()` 方法：

```
if (str1.equals(str2)) {
    System.out.println("兩個字串相同");
} else {
    System.out.println("兩個字串不同");
}
```

這次，因為 `str1` 和 `str2` 的內容是相同的，所以當我們使用 `.equals()` 方法來比較它們時，結果會顯示 "兩個字串相同"。

## Every object can be represented as String - toString()

在Java中，`toString()` 是一個來自 `Object` 類別的方法，每個Java object會繼承這個方法。`toString()` 方法的主要目的是返回一個代表該object字符串。默認實現會返回object的類別名稱，後面接著"@ "，然後是object的內存地址的無符號十六進制表示。然而，這種默認的表示對於大多數情況並不太有用。

因此，當我們創建自己的類別時，我們通常也會考慮override `toString()` 方法來提供對於該object更有意義的信息。例如，我們可能會讓 `toString()` 方法返回object的屬性的值，或者其他有助於理解該object的context。

`toString()` 方法的重要性主要體現在以下兩個方面：

1. 提供object的有用資訊：override的 `toString()` 方法可以提供有助於理解object狀態的資訊。這可以幫助我們在debug時更快地找到問題。
2. 提高程式的可讀性：當我們print一個object，Java會自動call `toString()` 方法。如果我們覆寫了 `toString()` 方法以返回有意義的資訊，那麼這些操作將產生更易讀的結果。

To sum, `toString()` 方法是Java object的一個重要特性，我們應該適當地覆寫它以提供object的有用資訊。

```
public class Student {
    private String name;
    private int age;

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

    @Override
    public String toString() {
        return "Student{" +
            "name='" + name + '\\\'' +
            ", age=" + age +
            '\'';
    }
}
```

在這個例子中，我們定義了一個Student類別並且覆寫了toString()方法。當我們創建一個Student物件並且呼叫toString()方法，它將返回一個表示該學生物件的字串。

在Java中，`toString()` 方法是一個來自 `Object` 類別的方法，這種方法對於呈現物件的內容非常有用。例如，我們可以使用HashMap類別來說明 `toString()` 的功能。

假設我們有一個HashMap來儲存學生的姓名與他們的成績：

```
HashMap<String, Integer> grades = new HashMap<String, Integer>();
grades.put("小明", 90);
grades.put("小華", 85);
grades.put("小李", 95);
```

如果我們直接嘗試印出這個HashMap，Java會自動call `toString()` 方法：

```
System.out.println(grades);
```

結果將會是：

```
{小明=90, 小華=85, 小李=95}
```

如上所示，`toString()` 方法幫助我們將一個複雜的物件（在這個例子中是一個 `HashMap`）轉換成一個易於理解的字串形式。這對於debug以及將物件的內容呈現給使用者非常有用。

## Array

在Java中，陣列（Array）也是一種物件（Object）。陣列是用來儲存同一類型的多個值的容器。我們可以在陣列中儲存基本資料類型（如int、char等）的值，也可以儲存物件的reference。

以下是一些陣列的範例：

```
// 創建一個整數陣列
int[] intArray = new int[5];

// 創建一個字串陣列
String[] strArray = new String[10];

// 給陣列賦值
intArray[0] = 1;
strArray[0] = "Hello";

// 讀取陣列中的值
int firstInt = intArray[0];
String firstStr = strArray[0];

// 陣列長度
int length = intArray.length;
```

在以上的範例中，首先創建了一個長度為5的整數陣列和一個長度為10的字串陣列。然後，將值賦給陣列中的元素。接著，讀取陣列中的元素值。最後，獲取陣列的長度。

在Java中，除了獲取陣列長度（length）外，還有一些其他重要的陣列操作，包括：

1. iterate陣列元素：可以使用for迴圈或者foreach迴圈來迭代陣列中的所有元素。

```
int[] intArray = {1, 2, 3, 4, 5};
// 使用for迴圈
```

```

for (int i = 0; i < intArray.length; i++) {
    System.out.println(intArray[i]);
}
// 使用foreach迴圈
for (int num : intArray) {
    System.out.println(num);
}

```

1. 多維陣列：在Java中，可以創建多維陣列，例如二維陣列。

```

int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

```

1. 陣列的複製：可以使用System.arraycopy()方法或者clone()方法來複製陣列。

```

int[] originalArray = {1, 2, 3, 4, 5};
int[] copiedArray = new int[5];

// 使用System.arraycopy()方法
System.arraycopy(originalArray, 0, copiedArray, 0, originalArray.length);

// 使用clone()方法
int[] clonedArray = originalArray.clone();

```

1. 陣列排序：可以使用Arrays.sort()方法來對陣列進行排序。

```

int[] unsortedArray = {5, 3, 2, 1, 4};
Arrays.sort(unsortedArray);
// unsortedArray現在為 {1, 2, 3, 4, 5}

```

## 匿名陣列Anonymous Array

在Java中，匿名陣列是沒有名稱的陣列，這意味著在創建它的時候不會給它指定一個變數名。匿名陣列可以在需要陣列但不需要再次使用該陣列的場合下使用，這樣可以

簡化code。一個常見的使用場景是在呼叫方法時直接傳遞一個陣列作為參數，而不打算在之後再使用這個陣列。

以下是一個簡單的範例，說明如何在Java中創建並使用匿名陣列：

```
// 定義一個方法，這個方法接受一個整數陣列作為參數並打印出所有元素
public class Main {
    public static void printArray(int[] array) {
        for (int i : array) {
            System.out.println(i);
        }
    }

    public static void main(String[] args) {
        // 直接在呼叫printArray方法時創建並傳遞一個匿名陣列
        printArray(new int[]{1, 2, 3, 4, 5});

        // 這裡傳遞的new int[]{1, 2, 3, 4, 5}就是一個匿名陣列，
        // 它在被創建後立即作為參數傳遞給printArray方法，而不會被賦予一個名稱。
    }
}
```

在上述範例中，我們定義了一個 `printArray` 方法，它接受一個整數型態的陣列作為參數。在 `main` 方法中，我們直接在呼叫 `printArray` 時創建了一個匿名陣列 `new int[]{1, 2, 3, 4, 5}`，並將它作為參數傳遞給了 `printArray` 方法。這樣，我們就可以在不需要給這個陣列一個明確名稱的情況下使用它，這對於只需要臨時使用陣列的場合非常有用。

## Java IO

Java.io套件提供了一系列的類別，用於讀取和寫入數據。這個套件包含了代表輸入/輸出錯誤的類別，以及用於數據讀取和寫入的串流（Stream）類別。



串流（Stream）在Java中是一種data的輸入輸出機制。在許多情況下，我們需要從一個來源讀取數據，或者將數據寫入到某個目的地。這些來源和目的地可以是檔案、記憶體結構、網路連線等。串流提供了一種抽象化的機制，能夠把這些不同的來源和目的地抽象化為統一的介面。

- **InputStream和OutputStream**：這是所有其他輸入/輸出位元byte(8bits)串流類別的基礎。InputStream是用於讀取數據的，而OutputStream則是用於寫入數據的。我們也可以說這是一種Byte Streams。
- **FileReader和FileWriter**：這兩個類別用於讀取和寫入為基礎的字元char(16bits)串流文件。這是最常用的，因為我們通常處理的數據是字元數據。我們也可以說這是一種Character Streams。
- **BufferedReader和BufferedWriter**：這兩個類別用於創建緩衝區，這可以提高讀寫操作的效率。這是一種Character Streams的實現。



在Java IO中，`InputStream` 和 `OutputStream` 主要用於讀取和寫入二進位數據，他們以位元組為單位進行操作。這意味著，他們可以處理所有類型的數據，包括文字、圖像、音頻等。

相反地，`Reader` 和 `Writer` 專門用於讀取和寫入文字數據，他們以字元為單位進行操作。這兩種類別在處理Unicode、UTF-8等多位元組字元集時，能夠提供比 `InputStream` 和 `OutputStream` 更為方便的操作。例如，它們可以自動處理字符編碼和解碼，使我們能夠更直觀地處理文字數據。

## FileReader、BufferedReader與InputStreamReader的串接方式

### FileReader

`FileReader` 是一種用於讀取字元文件的方便類別。以下是一個使用 `FileReader` 讀取文件的範例：

```
try {
    FileReader reader = new FileReader("test.txt");
    int character;

    while ((character = reader.read()) != -1) {
        System.out.print((char) character);
    }
    reader.close();

} catch (IOException e) {
```

```
e.printStackTrace();  
}
```

在這個範例中，我們首先創建了一個 `FileReader` 物件並指定要讀取的文件。然後，我們使用 `read()` 方法讀取文件中的每一個字元，直到達到文件的結尾（在這種情況下，`read()` 方法將返回 -1）。

## BufferedReader

`BufferedReader` 是一個包裝其他 `Reader` 的類別，並增加了緩衝功能，這可以提高讀取效能。以下是一個使用 `BufferedReader` 讀取文件的範例：

```
try {  
    FileReader reader = new FileReader("test.txt");  
    BufferedReader bufferedReader = new BufferedReader(read  
er);  
  
    String line;  
    while ((line = bufferedReader.readLine()) != null) {  
        System.out.println(line);  
    }  
    bufferedReader.close();  
  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

在這個範例中，我們首先創建了一個 `FileReader` 物件，然後將它包裝在一個 `BufferedReader` 物件中。然後，我們使用 `BufferedReader` 的 `readLine()` 方法讀取文件中的每一行，直到達到文件的結尾（在這種情況下，`readLine()` 方法將返回 `null`）。

## InputStreamReader

`InputStreamReader` 是一個橋接位元組流和字元串流的橋樑，它讀取位元組，並使用指定的編碼將其解碼為字元。它使用的字元集可以由名稱指定，也可以被明確指定，或者可以接受平台的默認字元集。以下是一個如何使用 `InputStreamReader` 讀取文件的範例：

```
try {  
    FileInputStream fileStream = new FileInputStream("test.
```



```

txt");
    InputStreamReader inputReader = new InputStreamReader(f
ileStream, "UTF-8");
    BufferedReader bufferedReader = new BufferedReader(inpu
tReader);

    String line;
    while ((line = bufferedReader.readLine()) != null) {
        System.out.println(line);
    }
    bufferedReader.close();

} catch (IOException e) {
    e.printStackTrace();
}

```

在這個範例中，我們首先創建了一個 `FileInputStream` 物件，然後用一個指定字元編碼集(UTF-8)的 `InputStreamReader` 將其包裝起來。然後，我們將 `InputStreamReader` 包裝在一個 `BufferedReader` 物件中。然後，我們使用 `BufferedReader` 的 `readLine()` 方法讀取文件中的每一行，直到達到文件的結尾（在這種情況下，`readLine()` 方法將返回 `null`）。

```

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class BufferedReaderInputStreamReaderExample {
    public static void main(String[] args) {
        try {
            BufferedReader bRer = new BufferedReader(
                new InputStreamReader(
                    new FileInputStream("exam
ple.txt"), "UTF-8"));
            String line;
            while ((line = bRer.readLine()) != null) {
                System.out.println(line);
            }
        }
    }
}

```

```

        bRer.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## Summary

- 使用 `FileReader` 對於簡單的文件讀取任務來說足夠了，但它不支持指定字元集 (UTF-8)。
- `BufferedReader` 提供了一種高效讀取文本文件的方式，特別是通過 `readLine()` 方法按行讀取。
- `InputStreamReader` 提供了從byte stream到char stream的轉換，並允許指定字元集，它通常與 `BufferedReader` 串接，以提供緩衝和按行讀取的功能。

## Scanner介紹

`Scanner` 是 `java.util` 套件中的一個類別（不是`java.io`喔），它用於獲取來自各種輸入源的基本類型和字符串。這個類別非常方便，因為它可以解析基本類型的字元串表示形式，例如整數、浮點數、布爾值等。此外，`Scanner` 也可以使用正規表示式來分割輸入。

以下是一個簡單的範例，說明如何使用 `Scanner` 來讀取使用者的輸入：

```

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("請輸入你的名字：");
        String name = scanner.nextLine();

        System.out.println("請輸入你的年齡：");
        int age = scanner.nextInt();

        System.out.println("嗨，" + name + "，你是 " + age +
            " 歲。");
    }
}

```

```

    }
}

```

在這個範例中，我們首先創建了一個 `Scanner` 物件，並將 `System.in`（標準輸入，通常是鍵盤）作為輸入源。然後，我們使用 `nextLine` 方法來讀取一行文本（即直到輸入換行符號為止）。接著，我們使用 `nextInt` 方法來讀取一個整數。最後，我們將讀取到的名字和年齡輸出到控制台。

以下是一個使用 `Scanner` 和 regular expression 式來分割輸入的範例：

```

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner("Apple,Banana,Cherry");
        scanner.useDelimiter(",");

        while(scanner.hasNext()){
            System.out.println(scanner.next());
        }

        scanner.close();
    }
}

```

在這個範例中，我們首先創建了一個 `Scanner` 物件，並將一個包含多個單詞的字串作為輸入源。然後，我們使用 `useDelimiter` 方法來設定分割輸入的分隔符號，此處我們將分隔符號設定為逗號。接著，我們使用 `hasNext` 和 `next` 方法來逐一讀取並輸出分割後的單詞。

以下是另一個 regular expression 的例子：

```

import java.util.Scanner;

public class ComplexRegexExample {
    public static void main(String[] args) {
        // 假設我們有一個包含單詞和數字，以逗號、分號或空格分隔的字串
        String input = "apple, banana; orange 123 grape,45;

```

```

mango 78";

        // 使用Scanner讀取這個字符串，並使用複雜的正則表達式作為分
        隔符
        // 正則表達式解釋：[,; ]+ 表示一個或多個逗號、分號或空格
        Scanner scanner = new Scanner(input).useDelimiter("[,; ]+");

        // 循環讀取並打印每個單詞或數字
        while (scanner.hasNext()) {
            System.out.println(scanner.next());
        }

        // 關閉Scanner
        scanner.close();
    }
}

```

在這個例子中，我們使用的正則表達式是 `"[,; ]+"`。這個regex匹配一個或多個逗號（`,`）、分號（`;`）或空格（）。這意味著無論這些分隔符出現單獨還是連續出現，`Scanner` 都將它們視為分隔符來分割輸入字符串。

`Scanner` 可以處理更複雜的輸入情況，其中可能包含不同類型的分隔符號。通過使用 regex 作為 `useDelimiter` 方法的參數，我們可以靈活地指定如何分割輸入字符串，這使得 `Scanner` 成為處理和解析多種格式輸入的強大工具。

## Java Serialization

### Java 序列化 (Serialization)

Java 序列化是一種將物件的狀態轉換為位元組流的機制，從而可以將其永久保存在硬碟上，或者將其通過網路傳輸到任何一台具有JVM的機器上。當其他的程式獲取位元組流，它可以將其反序列化，從而得到原始物件資訊。Java中，object的序列化主要透過 `java.io.Serializable` 介面來實現。

以下是一個序列化的範例：

```

import java.io.*;

public class Student implements Serializable {

```

```

private String name;
private int age;

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

public static void main(String[] args) {
    Student student = new Student("小明", 20);

    // 序列化
    try {
        FileOutputStream fos = new FileOutputStream("student.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(student);
        oos.close();
        fos.close();
        System.out.println("序列化成功!");
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }

    // 反序列化
    try {
        FileInputStream fis = new FileInputStream("student.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Student deserializedStudent = (Student) ois.readObject();
        ois.close();
        fis.close();
        System.out.println("反序列化成功!");
        System.out.println("學生姓名：" + deserializedStudent.name);
    }
}

```

```

        System.out.println("學生年齡：" + deserializedStudent.age);
    } catch (IOException ioe) {
        ioe.printStackTrace();
        return;
    } catch (ClassNotFoundException c) {
        System.out.println("Student類別未找到");
        c.printStackTrace();
        return;
    }
}
}
}

```

在這個範例中，我們首先創建了一個Student物件，並將其序列化到一個名為"student.ser"的檔案中。然後，我們從該檔案中反序列化該物件，並將反序列化後的物件的資訊印出。

## Java 序列化的優勢和利用場景 - 搜尋索引

Java 序列化的其中一個主要優勢是，能夠將物件的狀態保存並在需要的時候重建。這在建立搜尋索引的情境下顯得格外有用。

當我們要建立搜尋索引時，需要將大量的數據（如文件、網頁等）的內容和該數據的索引關聯起來。這樣，當用戶搜索特定的詞句時，系統能夠快速找到包含這些詞句的數據。然而，如果每次搜索都需要讀取並處理全部的數據，則會消耗大量的時間和資源。

這時，Java 序列化就可以發揮作用。我們可以將索引物件（包含了數據的索引和對應的數據）序列化並保存在硬碟上。然後，當用戶進行搜索時，我們只需要反序列化相應的索引物件，即可快速獲得對應的數據，而無需處理全部的數據。因此，Java 序列化能夠大大提高搜尋的效率。

以下是一個相關的 mermaid 圖：

```

sequenceDiagram
    participant User
    participant Search System
    participant Disk
    User->>Search System: Search "Michael Jordan"
    Search System->>Disk: Deserialize index objects

```

```
Disk-->>Search System: Return index objects
Search System->>User: Return search results
```

在這個流程圖中，用戶向搜尋系統發送搜索請求。搜尋系統接著從硬碟上反序列化相應的索引物件，並將搜索結果返回給用戶。這種方式能夠大大提高搜尋的速度和效率。

在使用 Java 序列化時，有一些需要注意的重要事項：

- **版本控制**：當一個類實現了 `Serializable` 介面，Java 運行環境會對該類別計算一個默認的 `serialVersionUID`，用於在序列化和反序列化時確保類別的版本一致。如果類別發生變動，`serialVersionUID` 可能會改變，這會導致反序列化失敗。我們可以在類別中明確地指定 `serialVersionUID` 以防止這種情況。

```
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    // ...
}
```

- **資料保護**：序列化物件可能會洩露私有資訊，因為序列化過程會將物件的全部狀態轉換為位元組流。對於需要保護的欄位，我們可以使用 `transient` 關鍵字來防止其被序列化。

```
public class Student implements Serializable {
    private String name;
    private transient String password; // 不會被序列化的欄位
    // ...
}
```

- **物件的一致性**：如果物件的狀態依賴於系統的其他部分，例如其他物件或系統時間等，那麼序列化可能會打破這種一致性。在這種情況下，我們可能需要在反序列化後手動恢復物件的狀態。

以下是一個包含 `HashMap` 和 `ArrayList` 的 Java 序列化範例：

```
import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;

public class SerializableExample implements Serializable {
```



```

private static final long serialVersionUID = 1L;
private HashMap<String, Integer> map;
private ArrayList<String> list;

public SerializableExample(HashMap<String, Integer> map, ArrayList<String> list) {
    this.map = map;
    this.list = list;
}

public static void main(String[] args) {
    HashMap<String, Integer> map = new HashMap<>();
    map.put("One", 1);
    map.put("Two", 2);
    map.put("Three", 3);

    ArrayList<String> list = new ArrayList<>();
    list.add("Apple");
    list.add("Banana");
    list.add("Cherry");

    SerializableExample example = new SerializableExample(map, list);

    // 序列化
    try {
        FileOutputStream fos = new FileOutputStream("example.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(example);
        oos.close();
        fos.close();
        System.out.println("序列化成功!");
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}

```

```

        // 反序列化
        try {
            FileInputStream fis = new FileInputStream("example.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);

            SerializableExample deserializedExample = (SerializableExample) ois.readObject();
            ois.close();
            fis.close();
            System.out.println("反序列化成功!");
            System.out.println("HashMap: " + deserializedExample.map);
            System.out.println("ArrayList: " + deserializedExample.list);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        } catch (ClassNotFoundException c) {
            System.out.println("SerializableExample類別未找到");
            c.printStackTrace();
        }
    }
}

```

在這個範例中，我們首先創建了一個 `SerializableExample` 物件，其內含一個 `HashMap` 和一個 `ArrayList`，並將其序列化到一個名為 "example.ser" 的檔案中。然後，我們從該檔案中反序列化該物件，並將反序列化後的物件的 `HashMap` 和 `ArrayList` 的內容印出。

要使用Java的序列化，必須確保所有涉及的物件都已實現 `Serializable` 介面。這包括任何被序列化物件內部所包含的所有物件。如果一個物件包含對其他物件的引用，那麼那些被引用的物件也必須實現 `Serializable` 介面，否則在執行序列化操作時會拋出 `NotSerializableException` 異常。這是因為序列化操作必須能夠遞歸地存儲一個物件及其所有的成員物件的狀態，以便將來可以恢復這個狀態。

除了已經提到的 `java.io` 和 `java.util` 套件，Java的標準函式庫還包含以下幾個重要的套件：

- `java.lang`：包含了 Java 程式語言的核心類別，如 `Object`、`String`、`Math`、`System` 等。

- `java.util.concurrent`：提供了一套強大的並行編程工具，如 `ExecutorService`、`ConcurrentHashMap` 等。
- `java.net`：包含了網路編程的相關類別，如 `URL`、`URLConnection`、`Socket` 等。
- `java.sql` 和 `javax.sql`：提供了與 SQL 數據庫互動的工具。
- `java.awt` 和 `javax.swing`：包含了建立圖形使用者介面的類別。
- `java.nio`：提供了非阻塞 I/O 的操作，這對於撰寫高效能的 I/O 程式非常有用。

## Java.net

`java.net` 套件提供了一系列的類別，用於實現網路相關的操作。這些類別主要可以分為以下幾種：

- **URL 和 URI**：這兩個類別用於表示統一資源定位符號（URL）和統一資源識別符號（URI）。它們提供了解析和操作URL和URI的方法。
- **InetAddress**：這個類別用於表示網際網路地址。它提供了獲取和操作網際網路地址的方法。
- **Socket 和 ServerSocket**：這兩個類別用於實現基於TCP協議的網路通訊。`Socket` 類別表示一個客戶端socket，而 `ServerSocket` 類別表示一個伺服器端socket。

以下是一個使用 `java.net` 套件的範例程式，這個程式創建了一個伺服器socket，並等待客戶端的連接：

```
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class SimpleServer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8888); // 創建一個伺服器套接字，並綁定到8888端口
            System.out.println("伺服器已啟動，等待客戶端連接...");

            while (true) {
```

```

        Socket socket = serverSocket.accept(); //
        等待客戶端的連接
        System.out.println("一個客戶端已連接");

        OutputStream outputStream = socket.getOutput
        tStream();
        PrintWriter printWriter = new PrintWriter(o
        utputStream);
        printWriter.write("你已經連接到伺服器");
        printWriter.flush();
        socket.shutdownOutput(); // 關閉輸出流

        socket.close(); // 關閉該連接
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

這個程式首先創建了一個 `ServerSocket` 伺服器套接字，並將其綁定到8888端口。然後，程式進入一個無窮迴圈，不斷地接受來自客戶端的連接請求。每當有一個客戶端連接到伺服器時，程式就會創建一個 `Socket` object，並通過該object的 `getOutputStream` 方法獲取輸出流，然後向客戶端發送一個歡迎訊息。最後，程式關閉與該客戶端的連接。

以下是一個連接到上述伺服器的客戶端程式範例：

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

public class SimpleClient {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("localhost", 8888);
            // 創建一個套接字並連接到伺服器

```

```

        // 獲取輸入流並讀取伺服器的響應
        BufferedReader bufferedReader = new BufferedRea
der(new InputStreamReader(socket.getInputStream()));
        String response = bufferedReader.readLine();
        System.out.println("伺服器響應： " + response);

        // 關閉資源
        bufferedReader.close();
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

這個客戶端程式首先創建一個 `Socket` 物件並連接到本地的8888端口（也就是我們的伺服器所在的位置）。然後，它獲取該socket的輸入流並讀取伺服器的響應。最後，程序關閉所有開啟的資源。

這兩個Java程式碼片段是一對簡單的TCP伺服器和客戶端。伺服器程式在8888端口監聽並接受客戶端的連接，當有客戶端連接時，它會發送一條歡迎訊息並關閉連接。客戶端程式則連接到本地的8888端口並讀取伺服器的message。

操作細節如下：

1. 首先，我們執行伺服器程式。程式創建一個ServerSocket object，並在8888端口上啟動監聽。然後，程式進入一個loop，等待客戶端的連接。
2. 接著，我們執行客戶端程式。程式創建一個Socket object，並試圖連接到本地的8888端口。
3. 當客戶端連接到伺服器時，伺服器程式建立一個Socket object，並透過該Socket object的輸出流向客戶端發送一條歡迎訊息。然後，伺服器關閉與該客戶端的連接。
4. 客戶端程式從Socket的輸入流讀取伺服器的響應，並將其印出。然後，客戶端關閉Socket。

在印出的結果中，我們應該可以看到伺服器印出"一個客戶端已連接"，並且客戶端印出"伺服器響應：你已經連接到伺服器"。



網路技術細節並非本堂課的重點，在這裡只是快速的介紹可以透過Java的套件，簡易、抽象去實現一個上層的服務操作，你不必理解太多網路底層，例如TCP、UDP的操作細節，只需要理解上層的使用介面的意義，即可創建一個具網路訊息交換功能的服務。

可以想像透過這樣子的方式，你就能很容易做出一個在使用者之間互丟傳訊息的服務，就像大家習慣使用的Line服務。

## Regular Expression

在Java中，我們可以使用java.util.regex套件來使用正則表達式(regular expression)。這個套件主要包含兩個類別：Pattern和Matcher。

- **Pattern**：Pattern類別代表一個編譯後的正則表達式。我們可以使用Pattern的靜態方法compile()來編譯一個正則表達式，得到一個Pattern物件。
- **Matcher**：Matcher類別可以對一個輸入序列進行匹配操作。我們可以使用Pattern物件的matcher()方法來獲得一個Matcher物件。

以下是一個使用正則表達式來驗證輸入的email是否有效的範例：

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexExample {
    public static void main(String[] args) {
        String regex = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\\\". [a-zA-Z]{2,6}$";
        String email = "example@email.com";

        // 編譯正則表達式
        Pattern pattern = Pattern.compile(regex);

        // 創建Matcher物件
        Matcher matcher = pattern.matcher(email);

        // 檢查是否匹配
        if (matcher.matches()) {
            System.out.println("Email is valid.");
        } else {
```

```

        System.out.println("Email is not valid.");
    }
}
}

```

在這個範例中，我們首先定義一個正則表達式，用於檢查一個字串是否符合email的格式。然後，我們使用Pattern的compile()方法來編譯這個正則表達式，得到一個Pattern物件。接著，我們使用Pattern物件的matcher()方法來創建一個Matcher物件，並將要檢查的字串傳遞給matcher()方法。最後，我們使用Matcher的matches()方法來檢查這個字串是否匹配我們的正則表達式。

在執行這個程式後，如果輸入的email符合我們的正則表達式，則程式將輸出"Email is valid."；否則，程式將輸出"Email is not valid."。



Regular Expression不是本門課的重點，但是他是一個常見且powerful的工具，你可以直接透過java的document了解這一個套件要怎麼使用，如：

<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

## Java.sql

`java.sql` 套件是Java提供的一套用於與SQL資料庫進行交互的工具。這個套件主要包含以下幾種類別：

- **Connection**：這個類別表示一個到資料庫的連接。它提供了創建 **Statement**、**PreparedStatement** 和 **CallableStatement** object的方法。
- **Statement**：這個類別用於執行靜態SQL語法並返回結果。
- **PreparedStatement**：這個類別是 **Statement** 的子類別，用於執行預編譯的SQL語句。與 **Statement** 不同，**PreparedStatement** 允許我們將SQL語句的參數化，從而提高效率並防止SQL注入攻擊。
- **ResultSet**：這個類別表示從資料庫查詢返回的結果集。它提供了獲取和操作結果集的方法。

以下是一個使用 `java.sql` 套件的範例程式，這個程式連接到一個MySQL資料庫，並執行一個SQL查詢：

```

import java.sql.Connection;
import java.sql.DriverManager;

```

```

import java.sql.ResultSet;
import java.sql.Statement;

public class DatabaseExample {
    public static void main(String[] args) {
        try {
            // 加載驅動
            Class.forName("com.mysql.jdbc.Driver");

            // 建立連接
            Connection conn = DriverManager.getConnection
("jdbc:mysql://localhost:3306/mydatabase", "username", "pas
sword");

            // 創建語句
            Statement stmt = conn.createStatement();

            // 執行查詢
            ResultSet rs = stmt.executeQuery("SELECT * FROM
students");

            // 處理結果集
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                int age = rs.getInt("age");
                System.out.println("ID: " + id + ", Name: "
+ name + ", Age: " + age);
            }

            // 關閉資源
            rs.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```



```
}  
}
```

這個程式首先加載MySQL的驅動，然後創建一個到資料庫的连接。接著，程式創建一個 `Statement` object，並使用該object執行一個SQL查詢。然後，程式獲取查詢返回的結果集，並將結果集中的每一條記錄印出。最後，程式關閉所有開啟的資源。

這個範例展示了如何使用 `java.sql` 套件與SQL資料庫進行交互，包括建立連接、執行SQL查詢、處理結果集，以及關閉資源等操作。透過這個套件，我們可以在Java程式中很方便地執行SQL查詢和更新操作。

**?** SQLite是一種嵌入式的關聯式資料庫管理系統。與其他資料庫系統如MySQL、PostgreSQL等不同的是，SQLite的運行不需要一個單獨的服務或伺服器，而且它將所有的數據存儲在一個單一的檔案中。這使得SQLite非常輕量級，並且非常適合一些需要輕量級資料庫解決方案的應用，例如嵌入式系統或者手機應用程式。儘管SQLite是一種輕量級的資料庫系統，但它仍然支援許多關聯式資料庫的主要特性，包括事務、子查詢和觸發器等。

## 安裝SQLite

SQLite 是一款輕量級的資料庫，不需要獨立的伺服器就能運行，並將所有數據存儲在單一檔案中。下面是安裝SQLite的步驟：

### Windows

1. 訪問SQLite官方網站的下載頁面：<https://www.sqlite.org/download.html>。
2. 在"Precompiled Binaries for Windows"部分下載合適版本。
3. 解壓縮下載的.zip檔案，將其中的三個.exe檔案移動到您希望的目錄中。
4. 將該目錄添加到您的PATH環境變數中。

### macOS

在macOS上，您可以使用Homebrew來安裝SQLite：

1. 開啟終端機。
2. 如果您還沒有安裝Homebrew，可以使用以下命令來安裝：

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

```
(echo; echo 'eval "$(/opt/homebrew/bin/brew shellenv)"') >>  
/Users/ktchuang/.zprofile  
eval "$(/opt/homebrew/bin/brew shellenv)"
```

1. 使用以下命令安裝SQLite：

```
brew install sqlite
```

## 使用SQLite

使用SQLite，您可以在命令列模式下操作資料庫，或者使用SQLite提供的API在程式碼中操作資料庫。

### 命令列模式

在命令列模式下，您可以使用 `sqlite3` 命令來啟動SQLite，並使用SQL語句來操作資料庫。例如，以下命令會創建一個名為 `mydatabase.db` 的新資料庫：

```
sqlite3 mydatabase.db
```

在SQLite的命令列模式下，您可以使用SQL語句來操作資料庫，例如創建表、插入數據、查詢數據等。

### 在程式碼中操作資料庫

如果您是在Java程式中使用SQLite，您需要下載SQLite的JDBC驅動。您可以在以下網址下載：

```
https://github.com/xerial/sqlite-jdbc
```

下載後，將JDBC驅動添加到您的Java專案的類路徑中，然後您就可以使用 `java.sql` 套件的API來操作資料庫了。以下是一個簡單的示例：

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class SQLiteExample {
    public static void main(String[] args) {
        Connection conn = null;
        try {
            // 註冊驅動並建立連接
            Class.forName("org.sqlite.JDBC");
            conn = DriverManager.getConnection("jdbc:sqlite:mydatabase.db");

            // 創建語句
            Statement stmt = conn.createStatement();

            // 執行查詢
            ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");

            // 處理結果集
            while (rs.next()) {
                // ...
            }

            rs.close();
            stmt.close();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (conn != null) {
                    conn.close();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

    }
}
}

```

在這個示例中，我們首先註冊SQLite的JDBC驅動，然後建立一個到 `mydatabase.db` 資料庫的連接。然後，我們創建一個 `Statement` object，並使用該object執行一個SQL查詢。最後，我們處理查詢返回的結果集，並關閉所有開啟的資源。

以下是一個使用 `java.sql` 套件連接到SQLite資料庫，並執行建立表格和查詢表格的範例：

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;

public class SQLiteExample {
    public static void main(String[] args) {
        Connection conn = null;
        try {
            // 加載SQLite的驅動
            Class.forName("org.sqlite.JDBC");

            // 建立到SQLite的連接
            conn = DriverManager.getConnection("jdbc:sqlite:
test.db");

            // 創建語句
            Statement stmt = conn.createStatement();

            // 執行建立表格的SQL語句
            String createTableSQL = "CREATE TABLE IF NOT EX
ISTS students (" +
                                   "id INTEGER PRIMARY KEY
AUTOINCREMENT, " +
                                   "name TEXT NOT NULL, "
+
                                   "age INTEGER NOT NUL
L)";

```

```

        stmt.execute(createTableSQL);

        // 執行查詢表格的SQL語句
        ResultSet rs = stmt.executeQuery("SELECT * FROM
students");

        // 處理結果集
        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            int age = rs.getInt("age");
            System.out.println("ID: " + id + ", Name: "
+ name + ", Age: " + age);
        }

        // 關閉資源
        rs.close();
        stmt.close();
        conn.close();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if(conn != null) {
                conn.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
}

```

在這個範例中，我們首先加載SQLite的驅動，然後建立一個到SQLite資料庫的連接。接著，創建一個 `Statement` 物件，並使用該物件執行建立表格的SQL語句。然後，再使用該物件執行查詢表格的SQL語句，並獲取查詢返回的結果集，將結果集中的每一條紀錄印出。最後，關閉所有開啟的資源。

## 讀取json檔案

JSON (JavaScript Object Notation) 是一種輕量級的數據交換格式，由兩種結構組成：名稱/值對的集合 (object) 和值的有序列表 (array)。名稱/值對由冒號分隔，名稱被雙引號包圍，名稱/值對之間用逗號分隔。物件由大括號括起來，而陣列由方括號括起來。

以下是一個簡單的 JSON 格式範例：

```
{
  "name": "John",
  "age": 30,
  "city": "New York",
  "pets": ["cat", "dog"]
}
```

在此範例中，我們有一個物件，其包含了四個名稱/值對。"name"、"age"和"city"的值分別為"John"、30和"New York"，而"pets"的值則是一個包含了兩個元素的陣列。

JSON 格式廣泛用於資料交換，因為它能夠輕易的被人類閱讀並被機器解析。

在 Java 中，我們可以使用 `org.json` 庫來讀取和處理 JSON 數據。以下是一個簡單的範例，該範例讀取一個 JSON 文件並輸出其內容：

```
import java.io.FileReader;
import java.io.IOException;
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

public class Main {
    public static void main(String[] args) {
        JSONParser parser = new JSONParser();

        try {
            Object obj = parser.parse(new FileReader("input.json"));
            JSONObject jsonObject = (JSONObject) obj;

            String name = (String) jsonObject.get("name");
```

```

        long age = (Long) jsonObject.get("age");
        String city = (String) jsonObject.get("city");

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("City: " + city);
    } catch (IOException | ParseException e) {
        e.printStackTrace();
    }
}
}
}

```

在這個範例中，我們首先創建一個 `JSONParser` 物件，然後使用 `parse()` 方法來解析 JSON 文件並返回一個物件。我們將返回的物件轉型為 `JSONObject`，然後使用 `get()` 方法來讀取 JSON 物件的各個欄位。最後，我們將讀取的欄位輸出到控制台。

請注意，你需要將 `org.json` library 添加到你的項目中才能使用上述程式。你可以使用 Maven 或 Gradle 來管理你的 code dependency，或者直接下載 jar 文件並將其加到你的 code space 中。這之後我們會再詳細說明。

## 身為程式設計師的 職業病問題

我媽說：  
「親愛的，幫忙去超市買1顆蘋果回來。  
如果他們有雞蛋的話，買6顆。」

最後我買了6顆蘋果回家。

她問：「你為什麼要買6顆蘋果?!」

我回答：「因為他們有雞蛋啊!!!!」