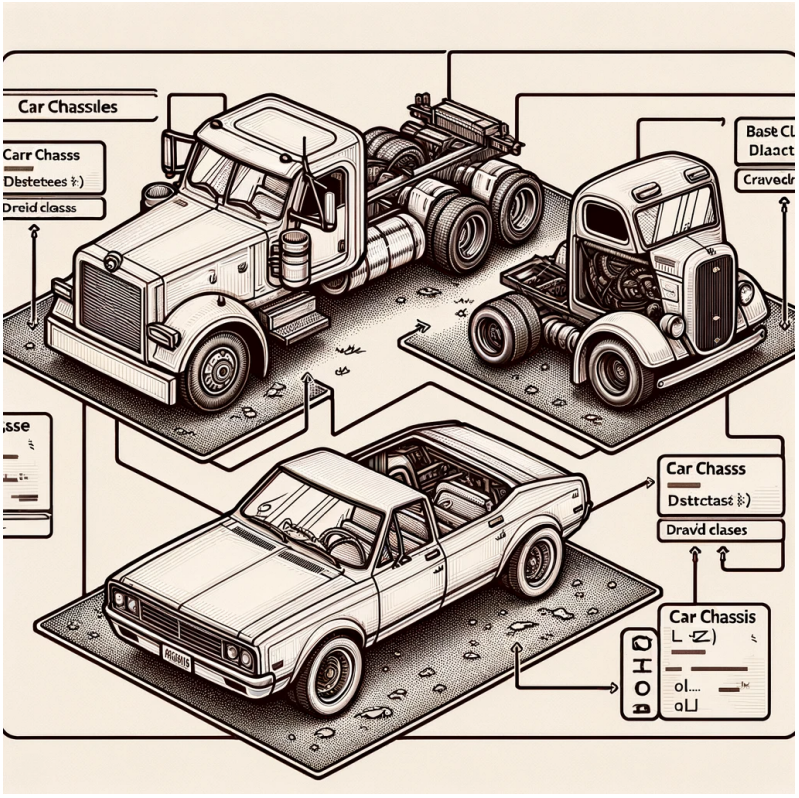
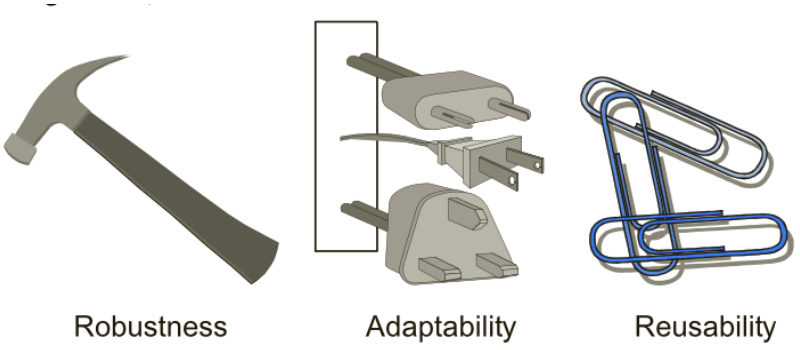
PD2 Lecture OOP - Introduction: Everything is Object



Generated by DALL.E

Goals of Object-Oriented Programming Design



Quoted from Data Structures and Algorithms in Java, 6/e

物件導向程式設計 (OOP) 的目標是創建擁有強健性（robustness）、可適應性（adaptability）和可重用性（reusability）的軟體：

- Robustness：是指軟體在面對預期之外的錯誤、輸入不確定性和異常條件時，能夠維持一定程度的功能運作的能力。物件導向設計通過封裝（Encapsulation）、繼承（Inheritance）和多型（Polymorphism）等機制，提高了軟體的錯誤處理能力和系統的整體穩定性。

> Encapsulation允許將Attribute（屬性）和Method（方法）綁定在一起，並隱藏內部實現的細節，從而減少外部干擾和意外錯誤。
> Inheritance使得可以通過擴展現有類別來添加或修改功能，而不影響其他部分。
> Polymorphism允許通過介面的一致性來處理不同類型的物件，從而提高代碼的靈活性和錯誤恢復能力。

每個優秀的程式設計師都希望開發出正確的軟體，也就是說一個程式對程式應用中預期的所有輸入產生正確的輸出。此外，我們希望軟體能夠強健，也就是說，能夠處理那些沒有明確為其應用定義的意外輸入。例如，如果一個程式期待一個正整數（可能代表一個商品的價格），但卻得到一個負整數，那麼程式應該能夠從這個錯誤中優雅地恢復過來。更重要的是，在***生命關鍵的應用***中，軟體錯誤可能導致人身傷害或生命損失，不強健的軟體可能會致命。這一點在1980年代後期的Therac-25輻射治療機事故中得到了驗證，這台機器在1985年至1987年間嚴重超劑量照射了六名患者，
其中一些人因放射性藥物過量引發的併發症而死亡。所有這六起事故都被追溯到軟體錯誤
wiki。

> 📖 Every good programmer wants to develop software that is correct, which means that a program produces the right output for all the anticipated inputs in the program's application. In addition, we want software to be ***robust***, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer (perhaps representing the price of an item) and instead is given a negative integer, then the program should be able to recover gracefully from this error. More importantly, in ***life-critical applications***, where a software error can lead to injury or loss of life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their radiation overdose. All six accidents were traced to software errors wiki.

- Adaptability：在 OOP 中，軟體的可適應性是指它能夠在不同的環境或情況下正確運行。這包括能夠處理不同的輸入、操作系統、硬體設備等。這通常是通過建立模塊化和可配置的(configurable)系統來實現的。

> 物件導向設計通過模塊化和解耦合decoupling的設計原則來提高軟體的適應性。類別的繼承和串接口的多型使得新的功能可以被輕鬆地添加進現有系統，而對其他部分的影響最小。此外，設計模式如策略模式Strategy Pattern、觀察者模式Observer Pattern等，提供了在不同情境下靈活應對變化的方法論。

像網頁瀏覽器和網路搜尋引擎這類的現代軟體應用程式，通常涉及大型程式且被使用多年。因此，軟體需要能隨著其環境中的變化條件進行隨時間演變。因此，優質軟體的另一個重要目標是實現***適應性***（也被稱為***演化性evolvability***）。與此概念相關的是***可攜性portability***，這是軟體在不同的硬體和作業系統平台上運行時只需進行最小改變的能力。使用Java撰寫軟體的一個優點是該語言本身提供innate的可攜性。

📖 Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software, therefore, needs to be able to evolve over time in response to changing conditions in its environment. Thus, another important goal of quality software is that it achieves ***adaptability*** (also called ***evolvability***). Related to this concept is ***portability***, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in Java is the portability provided by the language itself.

- Reusability：在 OOP 中，軟體的可重用性是指它的一部分（例如，類別或Method）可以在不同的程序中使用，而不需要進行大量的修改或重寫。這通常是通過建立generic的Interface、類別和方法，並使用繼承和組合來建立更複雜的系統來實現的。這降低了開發成本和時間，並提高了整體軟體質量。

> 物件導向的封裝特性使得具體功能的實現被包裹在類別中，易於在不同的項目中重用。通過繼承和組合，可以在不修改原有代碼的基礎上，擴展或自定義功能。此外，Design Pattern也為常見問題提供了可重用的解決方案，促進了程式碼的再利用。

與適應性相伴的是軟體需要具有可重用性，也就是說，同樣的程式碼應該能作為不同系統在各種應用中的組件。開發高品質的軟體可能是一項昂貴的事業，如果軟體的設計方式能使其在未來的應用中容易重用，那麼其成本可以得到一些抵銷。然而，這種重用應該謹慎進行，因為在Therac-25中，軟體錯誤的主要來源之一就是不當地重用了Therac-20的軟體（該軟體未設計用於Therac-25使用的硬體平台）。

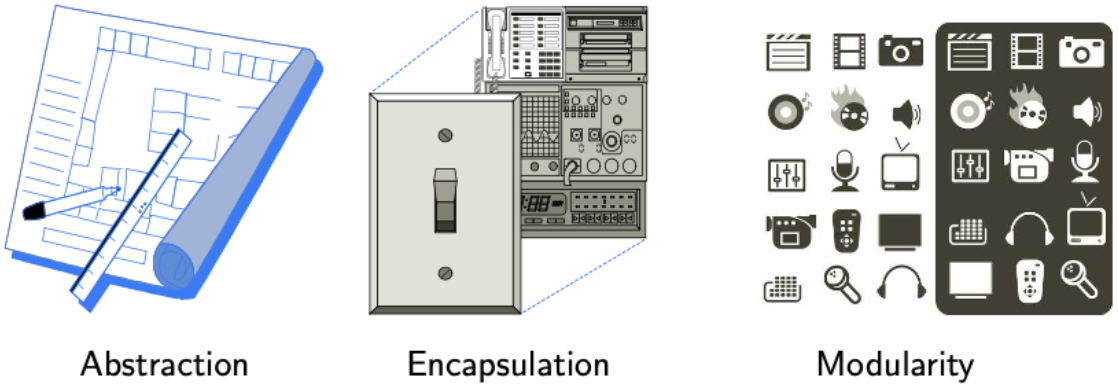
📖 Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications. Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications. Such reuse should be done with care. One of the major sources of software errors in the Therac-25 arose from the inappropriate reuse of software from the Therac-20 (which was not designed for the hardware platform used with the Theme-25).

Good Reference

- Design Goal and Principles of OOP

Design Goals and Principles of Object Oriented Programming - GeeksforGeeks
A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and practice/competitive programming/company interview Questions.
https://www.geeksforgeeks.org/design-goals-and-principles-of-object-oriented-programming/

Principles of Object-Oriented Programming Design



Quoted from Data Structures and Algorithms in Java, 6/e

為了達到Robustness、Adaptability、Reusability等目標，Object-Oriented程式設計必須善用三原則：

- Abstraction

  "抽象"的概念是將複雜系統純化為其最基本的部分。通常，描述系統的部分涉及命名它們並說明其功能。將抽象範疇應用於數據結構的設計，產生了"抽象數據類型"（ADTs）。一種ADT是數據結構的數學模型，它指定存儲的數據類型，它們支持的操作，以及操作的參數類型。ADT指定每個操作的"功能"，但不指定"如何"實現它。在Java中，ADT可以通過"Interface"來表示，這只是Method聲明的列表，其中每個方法都有一個空的主體。

  一個ADT由一個具體的數據結構實現，該結構在Java中由Class建模。Class定義了存儲的數據和由Class的實例支持的操作。此外，不同於Interface，Class在每個方法的主體中指定操作的"實現方式"。如果一個Java類別的方法包含接口中聲明的所有方法，從而為它們提供了主體，可說該Java類別"實現implement了一個Interface"。然而，一個Class可以有比Interface更多的Method。

  > 📖 The notion of ***abstraction*** is to distill a complicated system down to its most fundamental parts. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to ***abstract data types*** (ADTs). An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies ***what*** each operation does, but not ***how*** it does it. In Java, an ADT can be expressed by an ***interface***, which is simply a list of method declarations, where each method has an empty body.
  > An ADT is realized by a concrete data structure, which is modeled in Java by a ***class***. A class defines the data being stored and the operations supported by the objects that are instances of the class. Also, unlike interfaces, classes specify ***how*** the operations are performed in the body of each method. A Java class is said to ***implement an interface*** if its methods include all the methods declared in the interface, thus providing a body for them. However, a class can have more methods than those of the interface.

- Encapsulation

另一個物件導向設計的重要原則是***封裝***；軟體系統的不同組件不應洩露各自實現的內部細節。封裝的一個主要優點是，它賦予programmer自由實現一個組件的細節，而不用擔心其他programmer寫的元件的相關意涵。一個元件的開發者唯一要在意的是維護該組件的public interface，因為其他程序員將根據該介接方法寫對應的程式碼。封裝產生了健壯性和適應性。It allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

> 📖 Another important principle of object-oriented design is ***encapsulation***; different components of a software system should not reveal the internal details of their respective implementations. One of the main advantages of encapsulation is that it gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions. The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface. Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

- Modularity

現代軟體系統通常由幾個不同的元件組成，這些元件必須正確互動，以使整個系統能正常運作。保持這些互動的正確性需要將這些不同的元件適當組織。***模組化***指的是一種組織策略的原則，其中軟體系統的不同元件被劃分為獨立的功能單元。模組化大大提高了軟體的健壯性，因為在將單獨的元件整合到較大的軟體系統中之前，更容易測試和除錯這些組件。

> 📖 Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. ***Modularity*** refers to an organizing principle in which different components of a software system are divided into separate functional units. Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system.

Example of Inheritance I

```
---
title: Animal example
---
classDiagram
    note "From Duck till Zebra"
    Animal <|-- Duck
    note for Duck "can fly\ncan swim\ncan dive\ncan help in debugging"
    Animal <|-- Fish
```

```
    Animal <|-- Zebra
    Animal : +int age
    Animal : +String gender
    Animal: +isMammal()
    Animal: +mate()
    class Duck{
        +String beakColor
        +swim()
        +quack()
    }
    class Fish{
        -int sizeInFeet
        -canEat()
    }
    class Zebra{
        +bool is_wild
        +run()
    }
```

這個圖例子用於展示Java OOP的概念，特別是類別繼承、封裝和多態性的概念。在這個例子中，我們有一個基礎類別 `Animal`，以及三個從 `Animal` 類別繼承而來的子類別：`Duck`、`Fish` 和 `Zebra`。

類別繼承

類別繼承允許一個類別繼承另一個類別的屬性和方法。在這個例子中，`Duck`、`Fish` 和 `Zebra` 都是 `Animal` 的子類別，這意味著它們繼承了 `Animal` 類別的 `age`、`gender` 屬性和 `isMammal()`、`mate()` 方法。

封裝

封裝是OOP的一個核心概念，它意味著將對象的細節（狀態）隱藏起來，只通過公開的介面與外部世界互動。在 `Fish` 類別中，`sizeInFeet` 屬性被標記為私有（`-`），這表示只有 `Fish` 類別內部才能access這個屬性member。

多型

多型允許我們使用一個共同的介面來操作不同的對象。雖然這個例子中沒有直接展示多型，但你可以想象一個情況，在這種情況下，我們可以通過 `Animal` 類型的reference來存取 `Duck`、`Fish` 或 `Zebra` 的方法，而實際運行時會根據對象的實際類型來確定調用哪個類別的方法。

以下是根據這個例子的程式碼範例：

```
// 基礎類別Animal
abstract class Animal {
    int age;
    String gender;

    boolean isMammal() {
        // 根據具體動物類別實現的細節來決定
        return false;
    }
```

```
    void mate() {
        System.out.println("Find a partner.");
    }
}

// Duck類別
class Duck extends Animal {
    String beakColor;

    void swim() {
        System.out.println("Duck is swimming.");
    }

    void quack() {
        System.out.println("Duck says quack.");
    }
}

// Fish類別
class Fish extends Animal {
    private int sizeInFeet;

    private void canEat() {
        System.out.println("Fish is eating.");
    }
}

// Zebra類別
class Zebra extends Animal {
    boolean isWild;

    void run() {
        System.out.println("Zebra is running.");
    }
}

// 主類別
public class Main {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.swim();
        duck.quack();

        Zebra zebra = new Zebra();
        zebra.run();
```

```
        // 由於Fish的canEat方法是私有的，所以不能在Fish類的外部調用它
        // Fish fish = new Fish();
        // fish.canEat(); // 這行會產生編譯錯誤
    }
}
```

這個範例展示了如何在Java中定義基礎類別和子類別，以及如何實現封裝和繼承。透過這種方式，我們可以構建一個結構化且易於管理的程式能更加清晰且易於維護。

思考一個多型的運用case：

```
// 基礎類別Animal
abstract class Animal {
    int age;
    String gender;

    boolean isMammal() {
        // 根據具體動物類別實現的細節來決定
        return false;
    }

    void mate() {
        System.out.println("Find a partner.");
    }
}

// Duck類別
class Duck extends Animal {
    String beakColor;

    void swim() {
        System.out.println("Duck is swimming.");
    }

    void quack() {
        System.out.println("Duck says quack.");
    }
    void mate() {
        System.out.println("Find a Duck partner.");
    }
}

// Fish類別
class Fish extends Animal {
    private int sizeInFeet;

    private void canEat() {
```

```
        System.out.println("Fish is eating.");
    }
    void mate() {
        System.out.println("Find a Fish partner.");
    }
}

// Zebra類別
class Zebra extends Animal {
    boolean isWild;

    void run() {
        System.out.println("Zebra is running.");
    }
    void mate() {
        System.out.println("Find a Zebra partner.");
    }
}
// 主類別
public class Main {
    public static void main(String[] args) {
        Animal animal = new Duck();
        animal.mate();
    }
}
```

Example of Inheritance II

用汽車類別來解釋 OOP 的概念。在這個例子中，"汽車"是一個類別，這個類別擁有屬性，如顏色和品牌，還有方法，如啓動和停止。

首先，建立一個基本的"汽車"類別，這個類別具有一些基本的屬性，如品牌、顏色、和速度等，以及一些基本的方法，如啓動和停止。

```
public class Car {
    private String brand;
    private String color;
    private double speed;

    public Car(String brand, String color) {
        this.brand = brand;
        this.color = color;
        this.speed = 0;
    }

    public void start() {
        this.speed = 10;
```

```
    }

    public void stop() {
        this.speed = 0;
    }
}
```

接著，我們可以創建一個"跑車"類別，這個類別繼承自"汽車"類別，並添加一些新的特性，如最高速度和加速能力。

```
public class SportsCar extends Car {
    private double maxSpeed;
    private double acceleration;

    public SportsCar(String brand, String color, double maxSpeed, double
acceleration) {
        super(brand, color);  //呼叫parent constructor
        this.maxSpeed = maxSpeed;
        this.acceleration = acceleration;
    }

    public void accelerate() {
        this.speed += this.acceleration;
        if (this.speed > this.maxSpeed) {
            this.speed = this.maxSpeed;
        }
    }
}
```

透過這個例子，我們可以看到 OOP 的基本概念，包括封裝、繼承和多型。我們將汽車的基本特性封裝在"汽車"類別中，並通過繼承創建了一個更具體的"跑車"類別。此外，由於"跑車"類別繼承自"汽車"類別，我們可以將"跑車"object視為一個"汽車"object，這就是多型的概念。

在物件導向概念中，繼承是一種機制，其中一個類別（子類別）可以繼承另一個類別（父類別）的特性和方法。使用繼承，我們可以創建一個通用的類別，然後通過擴展這個類別來創建更具體的類別。這樣不僅可以重用代碼，還可以實現多型。

以下是將BMW、Tesla、Toyota作為子類別，繼承至Car類別的一個例子：

```
classDiagram
    class Car {
      +String brand
      +String color
      +int horsepower
      +accelerate()
      +stop()
      +turn(direction)
```

```
    }

    class BMW {
      +luxuryFeatures()
    }

    class Tesla {
      +electricMotor()
      +autopilot()
    }

    class Toyota {
      +reliability()
      +hybridEngine()
    }

    Car <|-- BMW : inherits
    Car <|-- Tesla : inherits
    Car <|-- Toyota : inherits

    Car : 🚗
    BMW : 🇩🇪
    Tesla : 🔋
    Toyota : 🌍
```

在這個例子中：

- `Car` 是一個基礎類別，提供了所有車子共有的基本屬性和方法。
- `BMW`、`Tesla`、`Toyota` 是從 `Car` 類別繼承而來的子類別。每個子類別都可以擁有自己獨特的方法，例如：
  - `BMW`（🇩🇪）類別可能有一個 `luxuryFeatures()` 方法，突出其豪華特性。
  - `Tesla`（🔋）類別擁有 `electricMotor()` 和 `autopilot()` 方法，反映其電動汽車和自動駕駛技術。
  - `Toyota`（🌍）類別包含 `reliability()` 和 `hybridEngine()` 方法，突出其可靠性和混合動力技術。

這樣的設計使得每個子類別都繼承了 `Car` 類別的共通特性，同時又能通過添加新的方法來展現各自的獨特性，實現重用性和擴展性。

Example of Encapsulation

封裝是OOP的一個核心概念，它涉及將數據（屬性）和代碼（方法）綁定到一個單元中——即類別裡，並對外界隱藏對象的具體實現細節。這樣做的目的是減少系統的複雜性並增加其可重用性。

為了擴展上文提到的Car類別，我們可以在Car中包含更多的物件，比如Engine和Wheel。以下是一個簡化的Java例子，展示了如何在Car類別中封裝Engine和Wheel類別的物件

先定義Engine和Wheel類別：

```
class Engine {
    private String type;

    public Engine(String type) {
        this.type = type;
    }

    public String getType() {
        return type;
    }
}

class Wheel {
    private int size;

    public Wheel(int size) {
        this.size = size;
    }

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

然後在Car類別中封裝這些物件：

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

    public Car(String engineType, int wheelSize) {
        this.engine = new Engine(engineType);
        this.wheels = new Wheel[4]; // 假設每輛車有4個輪子
        for (int i = 0; i < wheels.length; i++) {
            wheels[i] = new Wheel(wheelSize);
        }
    }

    // 方法來獲取引擎類型和輪子大小的信息
    public String getEngineType() {
        return engine.getType();
    }

    public int getWheelSize() {
        return wheels[0].getSize();
```

```
    }
}
```

```
classDiagram
    class Car {
      -Engine engine 🛢️
      -Wheel[] wheels 🛞
      +Car(engineType, wheelSize) 🚗
      +getEngineType() String 🔍
      +getWheelSize() int 🔍
    }
    class Engine {
      -String type 🏷️
      +Engine(type) 🛠️
      +getType() String 🔍
    }
    class Wheel {
      -int size 🔢
      +Wheel(size) 🛠️
      +getSize() int 🔍
    }
    Car *-- Engine : contains 🤝
    Car *-- Wheel : contains 🤝
```

這個圖表示Car類別包含了Engine和Wheel物件，展示了封裝的概念，即Car類別內部管理這些組件的創建和相關數據，對外部隱藏了這些實現細節。通過封裝，Car類的使用者不需要知道Engine和Wheel的具體實現細節，只需要通過Car類別提供的公共介面來使用汽車物件。

Recap - Object versus Class

在Java中，理解 `Object` 與 `Class` 的概念及其差異是理解OOP的基礎。以下是對這兩個概念的詳細說明以及相應的範例。

Class（類別）

- **定義**：`Class` 是一個藍圖或原型，它定義了創建物件（實例）時所需的狀態和行為。換句話說，它是物件的模板。
- **特點**：它包含數據成員（屬性）和方法（行為）。在OOP中，你可以通過 `class` 來創建任意多個具有相似屬性和行為的物件。

Object（物件）

- **定義**：`Object` 是根據 `Class` 藍圖創建的實例。每個物件都有其自己的狀態（屬性值）和行為（方法操作）。

- **特點**：即使兩個物件來自同一個`Class`，它們也可以擁有不同的狀態（即屬性值不同）。

假設我們有一個`Person`類別，它定義了人的一些基本屬性和功能。

定義一個Class（類別）

```
public class Person {
    String name; // 屬性
    int age;     // 屬性

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

    // Method to display info
    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

在上面的代碼中，`Person`類定義了兩個屬性（`name`和`age`）和一個方法（`displayInfo`），以及一個constructor來初始化這些屬性。

創建Object

接下來，我們可以使用`Person`類別來創建物件並對其進行操作。

```
public class TestPerson {
    public static void main(String[] args) {
        // Creating objects of Person class
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);

        // Calling method on objects
        person1.displayInfo(); // Output: Name: Alice, Age: 30
        person2.displayInfo(); // Output: Name: Bob, Age: 25
    }
}
```

在這個範例中，`person1`和`person2`是從`Person`類別創建的兩個不同的物件。每個物件都有自己的狀態（`name`和`age`的值）。這兩個物件都可以調用`displayInfo`方法來顯示它們自己的屬性資料。

```
classDiagram
    class Person {
      -String name 🏷️
      -int age 🔢
```

```
      +Person(name, age) 🚀
      +displayInfo() 📢
    }

    person1 : -String name = "Alice"
    person1 : -int age = 30
    person2 : -String name = "Bob"
    person2 : -int age = 25

    Person <|-- person1: Object
    Person <|-- person2: Object
```

簡結

- **Class**：是物件的藍圖，定義了創建物件時所需的屬性和方法。
- **Object**：是根據類別藍圖創建的實例，具有類別中定義的屬性和方法的具體實現。

Example of Polymorphism

在物件導向程式設計（OOP）中，多型是一種概念，允許我們以統一的界面來操作不同的資料類型。更具體地說，多型使得我們可以用同一個在父類別中的method，呼叫不同子類別中不同實現的method。

種類

1. **編譯時多型**（靜態多型）：透過方法重載（Method Overloading）。
   - 指的是在同一個類別中可以有多個同名方法，但它們的參數列表必須不同（參數類型、個數或者參數的順序不同）。
   - Method Overloading可讓同一個方法名稱可以根據不同的參數列表執行不同的任務。
2. **運行時多型**（動態多型）：透過方法覆寫（Method Overriding）。
   - 指的是子類別中定義了一個與父類別中具有相同名稱和參數列表的方法。在這種情況下，子類別的方法將在執行時取代父類別的同名方法。
   - Method Overriding可讓子類別提供一個屬於自己的特定實現方式，以替換繼承自父類的行為。

運行時多型示例(Override)

想像我們有一個`Animal`父類，它有一個`makeSound()`方法。`Dog`和`Cat`是`Animal`的子類，它們各自實現了`makeSound()`方法。

```
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    @Override
```

```
    void makeSound() {
        System.out.println("Woof");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow");
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myAnimal.makeSound(); // Prints "Some generic sound"
        myDog.makeSound(); // Prints "Woof"
        myCat.makeSound(); // Prints "Meow"
    }
}
```

```
classDiagram
    class Animal {
      <<abstract>> makeSound() 🗣️
    }
    class Dog {
      makeSound() 🐕
    }
    class Cat {
      makeSound() 🐈
    }

    Animal <|-- Dog : extends
    Animal <|-- Cat : extends
```

在這個圖中：

- `Animal`是一個帶有`makeSound()`方法的父類別。此方法在子類別中被override。
- `Dog`和`Cat`繼承自`Animal`並覆寫`makeSound()`方法，提供其對應的具體實現。
- 箭頭表示繼承關係，指向的是子類別從父類別繼承。

通過這個例子和圖解，我們可以看到多型如何使得我們能夠通過父類別的引用來調用在子類別中具有不同實現的方法，從而實現運行時的行為多樣性。這種機制使得程式碼更加靈活和可擴展。

How to call parent methods - super關鍵字

在Java物件導向程式設計（OOP）中，當子類別繼承父類別時，子類別可以覆蓋（override）父類別的方法。如果在子類別instance上調用這個方法，預設情況下會調用子類別中的版本。然而，有時候我們可能希望在子類別的instance上調用被覆蓋的父類別method。要實現這一點，可以使用以下兩種方式之一：

1. **在子類別的method中直接調用父類別的方法**：使用`super`關鍵字可以在子類別中調用父類別中被override的方法。
2. **顯式轉型（Casting）到父類別類型**：如果你有一個指向子類別object的reference，並且想要調用父類別中被覆蓋的方法，你需要將這個reference顯式轉型回父類別類型。

以下是使用`super`調用父類別方法的example：

```
class Parent {
    void display() {
        System.out.println("This is Parent class");
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("This is Child class");
    }

    void callParentDisplay() {
        super.display(); // 使用 super 調用父類的 display 方法
    }
}

public class TestCasting {
    public static void main(String[] args) {
        Child child = new Child();

        // 直接調用子類別的 display 方法
        child.display(); // 輸出: This is Child class

        // 通過子類別調用父類別的 display 方法
        child.callParentDisplay(); // 輸出: This is Parent class

    }
}
```

在這個例子中，`Child`類覆蓋了`Parent`類別的`display`方法。然而，通過`Child`類別的`callParentDisplay`方法和使用`super`關鍵字，我們可以在`Child`類別的instance上調用`Parent`類別的`display`方法。

Object Casting Example

以下是一個使用顯式轉型（Explicit Casting）的範例，我們將在這個範例中創建一個父類別 `Animal` 和兩個子類別 `Dog` 和 `Cat`。然後，我們將展示如何將 `Animal` 類型的reference顯式轉型為 `Dog` 或 `Cat` 類型的reference，以便訪問子類別自己特有的方法。

```
class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    public void makeSound() {
        System.out.println("Bark");
    }

    public void fetch() {
        System.out.println("Dog fetches");
    }
}

class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow");
    }

    public void scratch() {
        System.out.println("Cat scratches");
    }
}

public class TestCasting {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting, implicitly
        myAnimal.makeSound(); // Calls the overridden method in Dog

        // Downcasting, explicitly
        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal;
            myDog.fetch(); // Now we can access Dog-specific methods
        }

        // Attempting to use Cat-specific methods will require another in
stance
        myAnimal = new Cat(); // Upcasting, implicitly
        myAnimal.makeSound(); // Calls the overridden method in Cat

        // Downcasting, explicitly
```

```
        if (myAnimal instanceof Cat) {
            Cat myCat = (Cat) myAnimal;
            myCat.scratch(); // Now we can access Cat-specific methods
        }
    }
}
```

在這個範例中：

- 我們首先創建了`Animal`類別的reference`myAnimal`並將其指向一個新的`Dog`實例。這稱為向上轉型（Upcasting），是隱式的。
- 然後，我們使用`instanceof`檢查來確保`myAnimal`確實是一個`Dog`的實例，之後我們將其顯式轉型為`Dog`類型的reference，以便調用`Dog`特有的`fetch`方法。
- 同樣的過程也被用於將`myAnimal`reference指向一個`Cat`instance並顯式轉型以訪問`Cat`特有的`scratch`方法。

這顯示了如何通過顯式轉型來調用特定於子類別的方法，這些方法在父類別中是不能被使用的。顯式轉型在多態性和繼承中是一個強大的特性，但使用時需要小心，因為錯誤的轉型會導致`ClassCastException`。

instanceof關鍵字介紹

`instanceof`是一個關鍵字，用於檢查一個物件是否屬於特定的類別或其子類別。它也可以用來檢查一個物件是否實現了某個特定的interface。`instanceof`運算子return一個`boolean`值，如果物件是指定類別的instance，則返回`true`，否則返回`false`。

以下是一個使用`instanceof`關鍵字的範例程式：

```
class Animal { }

class Dog extends Animal { }

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Dog myDog = new Dog();

        System.out.println(myAnimal instanceof Animal); // Prints: true
        System.out.println(myDog instanceof Animal); // Prints: true
        System.out.println(myAnimal instanceof Dog); // Prints: false
    }
}
```

在這個範例中，`Dog`類別是`Animal`類別的子類別。我們創建了一個`Animal`物件`myAnimal`和一個`Dog`物件`myDog`。當我們檢查`myAnimal`是否是`Animal`的實例時，結果為`true`。

當我們檢查`myDog`是否是`Animal`的實例時，結果也為`true`，因為`Dog`是`Animal`的子類別。

但是，當我們檢查`myAnimal`是否是`Dog`的實例時，返回`false`，因為`myAnimal`的實際類型是`Animal`，不是`Dog`。

因此，`instanceof`關鍵字在處理多型時非常有用，可以讓我們在運行時確定物件的實際類別，並避免在轉型時出現`ClassCastException`。

Recap - Java語言對物件描述的框架

```
class ExtendedSampleClass extends SampleClass {   // 繼承 SampleClass
    public String attribute3;                              // 新增一個 publ
ic 字串屬性 attribute3

    public ExtendedSampleClass() {                         // 子類別的建構子
        super();                                           // 呼叫父類別的建
構子
        this.attribute3 = "Extended Hello";                // 初始化 attrib
ute3
    }

    @Override
    public void method1() {                                // 覆寫父類別的 m
ethod1 方法
        System.out.println("This is overridden method1 in ExtendedSampleC
lass.");
    }

    public void method3() {                                // 新增一個方法 m
ethod3
        System.out.println("This is method3 specific to ExtendedSampleCla
ss.");
    }
}

public class SampleClass {            // 宣告一個 public 類別 SampleClass
    private int attribute1;            // attribute1 是一個 private 整數屬
性
    public String attribute2;          // attribute2 是一個 public 字串屬性

    public SampleClass() {             // 這是一個 public 的建構子，建構子的
名字需與類別名稱相同
        this.attribute1 = 0;           // 在建構子內初始化 attribute1
        this.attribute2 = "Hello";     // 在建構子內初始化 attribute2
    }

    public void method1() {            // 宣告一個 public 的方法 method1
        System.out.println("This is method1.");
    }

    private void method2() {           // 宣告一個 private 的方法 method2
```

```
        System.out.println("This is method2.");
    }

    public static void main(String[] args) {  // main 方法是程式的入口點
        SampleClass sample = new SampleClass();              // 創建 Samp
leClass 的實例
        sample.method1();                                    // 呼叫 meth
od1
        System.out.println(sample.attribute2);               // 印出 attr
ibute2

        SampleClass extendedSample = new ExtendedSampleClass(); // 使用父
類型來參照子類別的實例
        extendedSample.method1();                            // 多型：呼叫
被覆寫的 method1
        System.out.println(extendedSample.attribute2);       // 可以訪問從
父類別繼承來的 attribute2
    // 注意：extendedSample.attribute3 不可訪問，因為它被宣告為 SampleClass 類
型的參考
    // 若要訪問 attribute3 或 method3，需要將 extendedSample 轉型為 ExtendedS
ampleClass

        if (extendedSample instanceof ExtendedSampleClass) {  // 檢查實例
的類型
            ExtendedSampleClass trueExtendedSample = (ExtendedSampleClas
s) extendedSample;
            System.out.println(trueExtendedSample.attribute3); // 現在可以
訪問 attribute3
            trueExtendedSample.method3();                     // 以及 me
thod3
        }
    }
}
```

在這個範例中:

- `public` 是一個存取修飾詞，表示該類別、屬性或方法可以被任何其他類別存取。
- `private` 也是一個存取修飾詞，表示該屬性或方法只能被所在的類別存取。
- `SampleClass` 是一個類別名稱，它包含了兩個屬性 `attribute1` 和 `attribute2`，以及兩個方法 `method1` 和 `method2`。
- `attribute1` 和 `method2` 是 private，所以它們只能在 `SampleClass` 類別內部被存取。
- `attribute2` 和 `method1` 是 public，所以它們可以在任何類別中被存取。
- `main` 方法是程式的進入點，JVM會在這裡開始執行程式。

💡 你能利用這個例子，解釋什麼是封裝、繼承、多型了嗎？

Constructor建構子

在Java中，建構子是一種特殊的方法，用於初始化新創建的物件。它的名稱必須與類別名稱完全相同，並且不返回任何類型（包括`void`）。當我們使用關鍵字 'new' 創建類別的新實例時，Java系統會自動調用相應的建構子來初始化該物件。

例如，以下程式碼定義了一個名為 'Person' 的類別，該類別具有兩個屬性 'name' 和 'age'，以及一個建構子，用於初始化這兩個屬性：

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

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

在這段程式碼中，當我們創建 'Person' 的新實例時，我們會調用該建構子，並將 'name' 和 'age' 的值作為參數傳入：

```
Person person = new Person("Alice", 25);
```

在這裡，'new Person("Alice", 25);' 的部分就是調用建構子的過程，創建了一個新的 'Person' 物件，並使用參數 "Alice" 和 25 初始化 'name' 和 'age' 屬性。

多個Constructor例子：

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

    // 第一個 constructor
    public Person(String name) {
        this.name = name;
        this.age = 0; // 預設年齡為 0
    }

    // 第二個 constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
```

```
    }
}
```

在這個範例中，`Person` 類別有兩個 constructor。第一個只接受一個 `name` 參數，並將 `age` 屬性設為預設值 0。第二個 constructor 接受 `name` 和 `age` 兩個參數，並將這兩個屬性都初始化。這是一個method overloading機制。

創建 `Person` 的新實例時，我們可以選擇調用哪一個 constructor：

```
Person person1 = new Person("Alice"); // 調用第一個 constructor，age 將設為預設值 0
Person person2 = new Person("Bob", 25); // 調用第二個 constructor，age 將設為 25
```

Recap - All Objects are 'reference'

在Java中，所有物件變數實際上都是以參照（reference）的形式存在。這意味著，當你創建一個物件變數時，你實際上是創建了一個參照，該參照指向物件在記憶體中的位置，而不是物件本身。因此，你可以將這個參照賦值給另一個物件變數，或者將其作為參數傳遞給方法，這些操作都不會創建新的物件，而只是增加了指向原始物件的參照數量。

這一點與基本數據類型（如int、double等）的行為不同，基本數據類型的變數在賦值或傳遞時會創建新的資料實體的copy。

因此，在使用物件變數時，我們必須注意這種參照行為，以避免因不正確的操作而產生意外的結果。

Recall - Call by value and Call by Reference

在Java中，方法的參數傳遞方式可分為 "Call by Value" 和 "Call by Reference"（或者說，都是Call by Value）。

Call by Value

"Call by Value" 意味著我們將一個變數的值傳遞給一個方法。這個方法可能會修改這個值，但是這個改變不會影響原來的變數，因為我們只是將值傳遞給方法，而不是變數本身。這種方式適用於基本數據類型，如 `int`、`double`、`boolean` 等。

例如：

```
void updateValue(int value) {
    value = 55;
}

public static void main(String[] args) {
    int value = 22;
    System.out.println("Before: " + value);  // Output: Before: 22
    updateValue(value);
    System.out.println("After: " + value);  // Output: After: 22
}
```

在這個例子中，儘管我們在 `updateValue()` 方法中修改了 `value` 的值，但是在方法外部，`value` 的值仍然沒有改變。這是因為我們是用 "Call by Value" 的方式傳遞參數，所以方法內的改變不會影響到原來的變量。

Call by Reference

"Call by Reference" 意味著我們將一個變數的Reference傳遞給方法。因此，如果方法修改了這個參照參照的物件，那麼這個改變會反映到原來的物件上。這種方式適用於物件。

例如：

```
class MyObject {
    int value;
}

void updateValue(MyObject obj) {
    obj.value = 55;
}

public static void main(String[] args) {
    MyObject obj = new MyObject();
    obj.value = 22;
    System.out.println("Before: " + obj.value);  // Output: Before: 22
    updateValue(obj);
    System.out.println("After: " + obj.value);  // Output: After: 55
}
```

在這個例子中，`updateValue()` 方法修改了 `obj.value` 的值，這個改變反映到了原來的物件 `obj` 上。這是因為我們是用 "Call by Reference" 的方式傳遞參數，所以方法內的改變會影響到原來的物件。

需要注意的是，雖然Java支持 "Call by Reference"，但是當我們嘗試改變reference本身（例如，使它參照到另一個物件）時，這個改變不會反映到原來的物件上。這是因為Java實際上是將參照的值（也就是物件的地址）傳遞給了方法，所以我們無法改變原來的reference。 — 也就是實質上，這也是一種Call by Value，只是那個'Value'是'Reference'的值。

以下是一個簡單的範例來說明這個概念：

```
class Dog {
    String name;

    Dog(String name) {
        this.name = name;
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return this.name;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Dog aDog = new Dog("Max");
        foo(aDog);

        // 當 foo 返回後，aDog 的名字已被改為Fifi
        System.out.println(aDog.getName()); // 輸出 "Fifi"，不是"Boxer"也不
是"Max"
    }

    public static void foo(Dog d) {
        d.getName().equals("Max"); // true
        // 改變 d 內的屬性
        d.setName("Fifi");
        d.getName().equals("Fifi"); // true
        // 但是，當我們嘗試將 d 參照到另一個物件時，這個改變不會反映到原來的物件上
        d = new Dog("Boxer");
        d.getName().equals("Boxer"); // true
    }
}
```

在這個範例中，當我們在 `foo` 方法中將 `d` 參照到一個新的 `Dog` 物件時，這個改變並不會反映到 `main` 方法中的 `aDog` 物件上。這是因為 Java 實際上是將 `aDog` 的參照值（也就是物件的地址）傳遞給了 `foo` 方法，所以我們無法改變 `aDog` 的參照。但我們在function中對 `d` 物件的修改，是會跟著改變的喔！

綜觀概念說明 - Java Dynamic Binding

定義

在Java中，動態綁定（Dynamic Binding）是一種在運行時確定物件方法調用的機制。這意味著當你調用一個物件的方法時，被調用的方法版本（子類別或父類別中的方法）是在程序運行時基於物件的實際類型決定的，而不是在編譯時。這是Java多型的核心。

特點

- **運行時決策**：系統在運行時才決定調用哪個類別的哪個方法。
- **與方法覆寫相關**：動態綁定主要與方法覆寫（Overriding）相關聯。
- **提升靈活性**：使得程式碼更加靈活，增加了程式的擴展性和可維護性。

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
```

```
class Dog extends Animal {
    @Override
    void eat() {
        System.out.println("Dog is eating");
    }
}

public class TestDynamicBinding {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.eat(); // Prints "Dog is eating"
    }
}
```

在這個例子中，雖然`myAnimal`的引用類型是`Animal`，但是實際指向的對象是`Dog`的一個實例。當調用`eat()`方法時，JVM在運行時確定`myAnimal`實際指向的是`Dog`對象，因此調用的是`Dog`類中覆寫的`eat()`方法，而不是`Animal`類中的版本。這就是動態綁定。

為了更好地理解動態綁定的概念，我們可以通過一個例子來表示`Animal`和`Dog`之間的關係，以及動態綁定的過程：

```
classDiagram
    class Animal {
      +eat() 🍽️
    }
    class Dog {
      +eat() 🍖
    }

    Animal <|-- Dog : extends
```

在這個圖中，`Animal`類別定義了一個`eat()`方法，而`Dog`類別繼承了`Animal`類別並覆寫了`eat()`方法。當使用`Animal`類別的reference調用`eat()`方法時，如果引用的實際object是`Dog`的實例，則根據動態綁定的原則，調用的將是`Dog`類中的`eat()`方法。

這個過程展示了Java中動態綁定的強大能力，它允許我們在不修改現有程式的情況下，通過擴展和覆寫方法來增加或改變程式的行為。

關鍵字this的用法

在Java中，`this`是一個關鍵字，用於參照當前物件，即正在被方法或建構子呼叫的物件。它有以下幾種用途：

- **參照物件的實例變數**：當方法的參數變數/區域變數與物件的Member變數名稱相同時，可以使用`this`來參照物件的實例Member變數。
- **呼叫物件的其他建構子**：在一個建構子裡，你可以使用`this()`來呼叫物件的其他建構子。

> 💡 在Java中，`this()`用於從一個建構子呼叫另一個建構子，而這個呼叫必須在建構子的第一行。這個規則的主要原因是，建構子的主要任務是初始化物件，並確保物件在被使用前已經完全建立。如果允許`this()`在建構子的其他地方進行呼叫，那麼物件的部分狀態可能已經被初始化，而其餘部分尚未初始化，這可能導致物件狀態的不一致。因此，為了確保物件的完整性，`this()`的呼叫必須在建構子的第一行。

- **作為當前物件reference的回傳**：你可以使用`this`來回傳當前物件的reference。

以下是一個示例程式碼：

```
public class Test {
    int x;

    // 建構子
    Test(int x) {
        this.x = x; // 使用 this 來參照物件的實例變數
    }

    void printX() {
        System.out.println("Value of x: " + this.x); // 使用 this 來參照物
件的實例變數
    }

    Test getThis() {
        return this; // 使用 this 來回傳當前物件的參照
    }

    public static void main(String[] args) {
        Test t = new Test(10);
        t.printX(); // 輸出 "Value of x: 10"
        System.out.println(t == t.getThis()); // 輸出 "true"
    }
}
```

在這個範例中，我們在建構子和`printX`方法中使用`this`來參照物件的實例變數`x`。我們也在`getThis`方法中使用`this`來回傳當前物件reference。

以下是另一個範例：

```
class Box {
    private int width, height, depth;

    // constructor，使用this區分實例變數和constructor參數
    public Box(int width, int height, int depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
```

```
    }

    // 設置尺寸的方法，使用this區分實例變數和方法參數
    public void setDimensions(int width, int height, int depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }

    // 使用this返回當前類的實例
    public Box getSelf() {
        return this;
    }

    // 顯示尺寸的方法
    public void displayDimensions() {
        System.out.println("Width: " + width + ", Height: " + height + ", D
    }
}

public class TestThis {
    public static void main(String[] args) {
        Box myBox = new Box(10, 20, 30);
        myBox.displayDimensions();  // 輸出: Width: 10, Height: 20, Depth: 3

        // 使用setDimensions方法重新設置尺寸
        myBox.setDimensions(50, 60, 70);
        myBox.displayDimensions();  // 輸出: Width: 50, Height: 60, Depth: 7

        // 使用this返回當前類的實例
        Box anotherBox = myBox.getSelf();
        anotherBox.displayDimensions();  // 輸出: Width: 50, Height: 60, Dep
    }
}
```

在這個範例中，我們可以看到幾種使用`this`的情形：

- 在`Box`類別的constructor中，我們使用`this`來區分參數名和類別的實例變數名。
- 在`setDimensions`方法中，同樣使用`this`來引用當前object的instance變數。
- 在`getSelf`方法中，通過`this`返回當前object的reference。

Interface簡介

在Java中，介面（Interface）是一種reference類型，它是完全抽象的類型，即它不能包含具體的實現方法（正確來說 - Java 8之後，Interface可以包含static method）。Interface是定義了某一批類別所

需要遵守的規範，它是這些類別的公共抽象方法的集合。當一個類別實現了某個Interface，它會被要求提供Interface中所有方法的具體實現（implementation）。

Interface的主要用途：

- 實現抽象：提供了一種機制，用於指定一組必須由實作此interface的類別來實現的方法。
- 實現多重繼承：Java不支持多重繼承（一個類別有多個父類別），但通過實現多個Interface，一個類別可以"類似"繼承了多個Interface的特性。

實現interface是透過`implements`關鍵字，讓一個類別去實現一個或多個interface。當一個類別使用`implements`關鍵字後，該類別必須提供該接口中所有抽象方法的具體實現內容。

如果一個類別要實現多個interface，可以使用逗號分隔每個接口的名稱：

```
class ClassName implements Interface1, Interface2, Interface3 {
    // 提供所有接口的方法實現
}
```

以下是一個簡單的Interface例子，我們定義了一個Interface`Vehicle`，它有一個方法`move()`。然後我們定義了兩個類別`Car`和`Bicycle`，它們都實現了`Vehicle`Interface，並提供了`move()`方法的具體實現(implementation)。

```
// 定義Interface Vehicle
interface Vehicle {
    // Interface中的抽象方法 move
    void move();
}

// Car 類別實現 Vehicle Interface
class Car implements Vehicle {
    // 實現 move 方法
    public void move() {
        System.out.println("Car is moving");
    }
}

// Bicycle 類別實現 Vehicle Interface
class Bicycle implements Vehicle {
    // 實現 move 方法
    public void move() {
        System.out.println("Bicycle is moving");
    }
}

public class TestInterface {
    public static void main(String[] args) {
        // 創建一個 Vehicle 類型的reference指向 Car object
        Vehicle myCar = new Car();
        myCar.move();  // 輸出: Car is moving
```

```
        // 創建一個 Vehicle 類型的reference指向 Bicycle object
        Vehicle myBicycle = new Bicycle();
        myBicycle.move();  // 輸出: Bicycle is moving
    }
}
```

在這個例子中，`Vehicle` Interface定義了`move()`方法，而`Car`和`Bicycle`類別都實現了這個接口。這表示它們都必須提供`move()`方法的implementation。這種機制允許我們使用Interface類型的reference（如`Vehicle`）來指向任何實現了該Interface的類別的實例，進而實現了多型機制。

使用interface來實作有幾個主要優點：

- **封裝性**：Interface可以將方法聲明(declaration)和implementation分開，這有助於保持程式碼的整潔和可讀性。
- **靈活性和可擴展性**：Interface提供了一種機制，讓開發者可以更改或增加類別的行為，而無需改變現有的類別。這增加了程式碼的靈活性和可擴展性。以下是一個使用Interface來增加類別行為的範例：

```
// 定義一個 Flyable Interface
interface Flyable {
    // Interface中的抽象方法 fly
    void fly();
}

// 定義一個 Bird 類別
class Bird {
    // Bird的行為
    void eat() {
        System.out.println("Bird is eating");
    }
}

// Penguin類別繼承Bird類別，並實現Flyable接口
class Penguin extends Bird implements Flyable {
    // 實現 fly 方法
    public void fly() {
        System.out.println("Penguin can fly");
    }
}

public class TestInterface {
    public static void main(String[] args) {
        // 創建一隻Penguin
        Penguin penguin = new Penguin();
        penguin.eat();  // 輸出: Bird is eating
        penguin.fly();  // 輸出: Penguin can fly
```

```
    }
}
```

在這個例子中，`Bird` 類別有一個 `eat()` 方法。然後我們定義了一個 `Flyable` Interface，並實現這個Interface來擴展 `Penguin` 類別的行為。這樣，Penguin類別就繼承了Bird類別的 `eat()` 方法，並增加了新的行為 `fly()`。這種機制讓我們可以在不改變 `Bird` 類別的情況下，為 `Penguin` 類別增加新的行為，增加了程式碼的靈活性和可擴展性。

- **多重繼承**：Java不支持多重繼承，即一個類別不能有多個父類別。但是，一個類別可以實現多個Interface，這類似於多重繼承，可以讓一個類別繼承多個Interface的特性。

```
// 定義一個 Runnable Interface
interface Runnable {
    // Interface中的抽象方法 run
    void run();
}

// 定義一個 Swimmable Interface
interface Swimmable {
    // Interface中的抽象方法 swim
    void swim();
}

// 定義一個 Robot 類別，同時實現 Runnable 和 Swimmable 兩個 Interface
class Robot implements Runnable, Swimmable {
    // 實現 run 方法
    public void run() {
        System.out.println("Robot is running");
    }

    // 實現 swim 方法
    public void swim() {
        System.out.println("Robot is swimming");
    }
}

public class TestInterface {
    public static void main(String[] args) {
        // 創建一個 Robot
        Robot myRobot = new Robot();
        myRobot.run();  // 輸出: Robot is running
        myRobot.swim();  // 輸出: Robot is swimming
    }
}
```

在這個例子中，`Robot` 類別同時實現了 `Runnable` 和 `Swimmable` 兩個Interface，因此必須提供 `run()` 和 `swim()` 兩個方法的具體實現。這顯示了一個類別如何透過實現多個Interface來達到類似多重繼承的效果。

- **多型支援**：一個類別可以實現多個Interface，並可以使用Interface類型的reference來指向實現了該Interface的任何類別的instance。這允許我們在不知道實際class的情況下，用一種通用的方式來處理物件，可以實現多型的支援。

```
// 定義一個 Interface Animal
interface Animal {
    // Interface中的抽象方法 makeSound
    void makeSound();
}

// Dog 類別實現 Animal Interface
class Dog implements Animal {
    // 實現 makeSound 方法
    public void makeSound() {
        System.out.println("Dog says: Woof Woof!");
    }
}

// Cat 類別實現 Animal Interface
class Cat implements Animal {
    // 實現 makeSound 方法
    public void makeSound() {
        System.out.println("Cat says: Meow Meow!");
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        // 創建一個 Animal 類型的reference指向 Dog object
        Animal myDog = new Dog();
        myDog.makeSound();  // 輸出: Dog says: Woof Woof!

        // 創建一個 Animal 類型的reference指向 Cat object
        Animal myCat = new Cat();
        myCat.makeSound();  // 輸出: Cat says: Meow Meow!
    }
}
```

在這個例子中，`Animal` Interface定義了`makeSound()`方法，而`Dog`和`Cat`類別都實現了這個接口。這表示它們都必須提供`makeSound()`方法的實現。這種機制允許我們使用Interface類型的reference（如`Animal`）來指向任何實現了該Interface的類別的實例，進而實現了多型機制。

其它的程式撰寫上的優點：

- **促進低耦合Low Coupling設計：**通過使用interface作為不同組件之間的介面，可以減少組件之間的依賴。這種低耦合設計使得系統的各個部分可以獨立地開發和測試，提高了系統的穩定性和靈活性。
- **促進團隊合作和開發效率：**在大型軟體開發中，interface可以作為不同開發隊伍之間的溝通介面。一個隊伍可以負責interface的定義，而其他隊伍可以同時進行interface的實現和使用，這樣可以並

行工作，提高開發效率。

- **支援安全隱藏實現細節：**interface只定義要暴露的方法，不涉及具體的實現細節，這有助於隱藏實現的複雜性，只向外界提供清晰、簡潔的interface，增加了程式碼的安全性和易用性

> ❓ 如果你要 `implements` 兩個interface，其中都需實作一個同名的method，會發生什麼事呢?

Interface Member Variable

在Java的interface中，您可以有成員變數，但是它們會自動默認為 `public`、 `static` 跟 `final` 的variable。換句話說，Interface的成員變數實際上是常數。因此，一旦它們被初始化，就不能更改它們的值。以下是一個範例：

```
interface MyInterface {
    // 這是一個接口中的成員變數，它實際上是一個常數
    int MY_CONSTANT = 10;
}

class MyClass implements MyInterface {
    public void printConstant() {
        System.out.println("The constant is: " + MY_CONSTANT);
    }
}

public class TestInterface {
    public static void main(String[] args) {
        MyClass myObject = new MyClass();
        myObject.printConstant();  // 輸出: The constant is: 10
    }
}
```

在這個例子中， `MyInterface` Interface有一個成員變數 `MY_CONSTANT`。 `MyClass` 類實現了該介面，並可以訪問該常數。然而，它不能改變 `MY_CONSTANT` 的值，因為它實際上真的是一個常數。

Abstract Class介紹

抽象類別（Abstract Class）是一種不能直接實例化（不可透過 `new` 來生成物件）的特殊類別。我們可以通過繼承抽象類別並實現其抽象方法來創建新的類別。抽象類別通常被用來作為所有具有共同特性的類別的基礎。是一種介於Interface（不能提供任何實作內容）與class之間的物件機制。

特點

- **抽象方法（abstraction method)**：抽象類別可以包含一個或多個抽象方法。抽象方法是一種只有聲明沒有實現的方法，它的實現由子類別提供。
- **實例方法(instance method)**：除了抽象方法外，抽象類別也可以包含實例方法。這些方法可以有實現，子類別可以選擇覆寫或直接使用它們。

- **不能直接實例化**：我們不能直接使用 `new` 關鍵字來創建抽象類別的實例。然而，我們可以使用抽象類別類型的reference來引用子類別的實例。

以下是一個抽象類別的範例：

```
// 定義一個抽象類別 Animal
abstract class Animal {
    // 抽象方法 sound
    abstract void sound();
}

// Dog 類別繼承 Animal 抽象類別
class Dog extends Animal {
    // 實現 sound 方法
    public void sound() {
        System.out.println("Dog says: Woof Woof!");
    }
}

public class TestAbstractClass {
    public static void main(String[] args) {
        // 創建一個 Dog
        Dog myDog = new Dog();
        myDog.sound();  // 輸出: Dog says: Woof Woof!
    }
}
```

在這個例子中，`Animal` 是一個抽象類別，它有一個抽象方法 `sound()`。`Dog` 類別繼承了 `Animal` 並提供了 `sound()` 方法的實現。這種機制允許我們定義公共的方法（如 `sound()`），並要求每個子類別提供自己的實現。這樣，我們就可以在不知道實際類別的情況下，通過一個 `Animal` 參照來調用 `sound()` 方法，這是多型的一種體現。

❓ 想想看，一個abstract class能不能被另一個abstract class繼承？

Inner Class

Inner Class是一種定義在另一個類別內部的類別。這種機制允許我們將一些相關的類別組織在一起，讓程式碼更加整潔且易於維護。內部類也可以訪問其外部類別的成員（包括private member data）。

以下是一個內部類的範例：

```
class OuterClass {
    private String myField = "Hello, World!";

    class InnerClass {
        void printMyField() {
            // 內部類可以訪問外部類的成員
```

```
            System.out.println(myField);
        }
    }

    void testInnerClass() {
        // 創建內部類的實例
        InnerClass myInner = new InnerClass();
        myInner.printMyField();
    }
}

public class TestInnerClass {
    public static void main(String[] args) {
        // 創建外部類的實例
        OuterClass myOuter = new OuterClass();
        myOuter.testInnerClass(); // 輸出: Hello, World!
    }
}
```

在這個範例中，`OuterClass` 是一個外部類別，它有一個內部類別 `InnerClass`。`InnerClass` 可以訪問 `OuterClass` 的私有成員 `myField`。

在 Java 中，內部類別（Inner Class）有權利讀取和修改外部類別（Outer Class）的所有成員。

如果內部類別和外部類別有同名的變數或方法，內部類別會默認訪問其自身的變數或方法。如果想要訪問外部類別的同名變數或方法，我們可以使用 `外部類別名.this.變數名` 或 `外部類別名.this.方法名()` 的方式來進行訪問。

```
class OuterClass {
    private String myField = "Hello, Outer World!";

    class InnerClass {
        private String myField = "Hello, Inner World!";

        void printMyField() {
            // 內部類別讀取自己的變數
            System.out.println(myField);

            // 內部類別讀取外部類別的變數
            System.out.println(OuterClass.this.myField);
        }
    }

    void testInnerClass() {
        // 創建內部類別的實例
        InnerClass myInner = new InnerClass();
        myInner.printMyField();
    }
```

```
}

public class TestInnerClass {
    public static void main(String[] args) {
        // 創建外部類的實例
        OuterClass myOuter = new OuterClass();
        myOuter.testInnerClass();
        // 輸出: Hello, Inner World!
        // 輸出: Hello, Outer World!
    }
}
```

在這個例子中，`OuterClass` 是一個外部類別，它有一個內部類別 `InnerClass`。`InnerClass` 和 `OuterClass` 都有一個同名的變數 `myField`。當內部類別想要訪問其自身的 `myField` 時，直接使用 `myField` 即可；當內部類別想要訪問外部類別的 `myField` 時，則需要使用 `OuterClass.this.myField`。

📌 在Java中，編譯後的Inner Class會產生一個獨立的 .class 檔案。該檔案的命名規則是 `外部類別名$內部類別名.class`。例如，如果有一個名為 `OuterClass` 的外部類別，內部有一個名為 `InnerClass` 的內部類別，則編譯後會產生一個名為 `OuterClass$InnerClass.class` 的檔案。

📌 現階段不鼓勵使用 - 但你可以問chatGPT什麼是java的anonymous class，以及Java 8延伸出的Lambda表達式。

final method介紹

在Java中，如果一個方法被定義為final，它有以下特性和限制：

- **不可被覆寫**: 一個被定義為final的方法不能被子類別覆寫（override）。這意味著如果你在父類別中定義了一個final方法，子類別不能提供這個方法的不同實現。
- **不可變性**: final方法的內容在定義之後不能更改。這就像是一個常數方法，你可以依賴它的行為永遠不會改變。

下面是一個範例：

```
public class ParentClass {
    // 定義一個 final 方法
    public final void finalMethod() {
        System.out.println("This is a final method.");
    }
}

public class ChildClass extends ParentClass {
    // 試圖覆寫 final 方法會引發編譯錯誤
    // public void finalMethod() {
    //     System.out.println("Trying to override a final method.");
    //}
```

```
}

public class Main {
    public static void main(String[] args) {
        ChildClass child = new ChildClass();
        child.finalMethod();  // 輸出: This is a final method.
    }
}
```

在這個範例中，`finalMethod()`被定義為final，所以它不能在`ChildClass`中被覆寫。這個特性在你需要確保方法的行為不被更改，或者在你需要防止在子類別中改變某些方法的行為時非常有用。

Java Package的簡單介紹

在Java中，package是用來將相關的類別和接口組織在一起的一種naming scope機制。使用package可以避免類別名稱的衝突，並可以更有效地管理程式碼。package提供了一種封裝機制，可以控制類別和成員的可見性。

在Java中，每個類別都屬於某個package。如果未明確指定package，則類別屬於無名稱的package（default package）。無名稱package的作用範圍僅限於當前的目錄，基本不推薦在正式的程式碼中使用無名稱package。

為了創建一個新的package，我們可以在程式碼的開頭使用`package`關鍵字，後面跟著package的名稱：

```
package com.example.myapp;
```

在這個例子中，我們創建了一個名為`com.example.myapp`的package。package名稱通常使用小寫字母，並且可以包含多個部分，各部分之間用點（.）分隔。為了避免名稱衝突，通常會使用公司網域的反向順序作為package的前字詞。

當我們需要在程式碼中使用其他package的類別時，我們需要使用`import`關鍵字來引用該類別。例如，如果我們想要使用`java.util.ArrayList`類別，我們需要在程式碼的開頭添加如下的import語句：

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

這樣，我們就可以在程式碼中直接使用`ArrayList`類別，而不需要每次都寫出完整的類別名稱`java.util.ArrayList`。

如果我們需要使用同一個package的多個類別，我們可以使用星號（*）來引用該package中的所有類別：

```
import java.util.*;
```

這樣，我們就可以在程式碼中直接使用`java.util` package中的所有類別。

以下是一個使用package的範例：

```
package com.example.myapp;
```

```
import java.util.ArrayList;

public class MyApp {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Hello");
        list.add("World");
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

在這個範例中，我們創建了一個 `com.example.myapp` 的package，並在其中定義了一個 `MyApp` 類別。我們使用 `import` 關鍵字來引用 `java.util.ArrayList` 類別，並在 `main` 方法中使用該類別來創建一個 `ArrayList` 物件。

目錄結構

在文件系統中，package的結構反映為目錄（資料夾）結構。例如，上述 `com.example.app` 中的 `HelloWorld` 類別，應該存儲在一個路徑為 `com/example/app/` 的目錄中。

Java的標準類別庫（Java Standard Class Library）是Java語言提供的一套標準的類別庫，其中包含了許多用於開發各種應用程式的工具與API。這些類別庫提供了一些基本的功能，例如檔案I/O、網路編程、數據結構、圖形介面、多執行緒編程等等。這樣可以讓開發者不需要從零開始實作這些基本功能，而是可以直接使用標準類別庫中提供的工具來完成這些任務。

Java 標準類別庫	描述
`java.lang`	提供了Java程式語言的核心類別，包含了 `Object`、`Math`、`String`、`Thread` 等等類別。
`java.util`	提供了一些實用工具類別與集合框架，例如 `ArrayList`、`HashMap`、`Date`、`Random` 等等類別。
`java.io`	提供了讀取與寫入資料的I/O類別，例如 `File`、`InputStream`、`OutputStream` 等等類別。
`java.net`	提供了網路程式設計的類別，例如 `URL`、`Socket` 等等類別。
`java.sql`	提供了與SQL資料庫連線和操作的類別。
`java.awt` 和 `javax.swing`	提供了建立圖形使用者介面的類別。

📖 Java Package的包裝對大型軟體開發十分重要，我們會在後頭才介紹怎麼規劃library的封裝跟階層的設計

Recap - Variable Visibility

- **Public**: 這是最寬泛的存取級別。被定義為public的類別、方法或者變數可以被任何其他的類別訪問。

- **Private**: 這是最限制的存取級別。被定義為private的方法或者變數僅能被同一類別中的其他方法訪問，其他類別是無法訪問的。
- **Protected**: 被定義為protected的方法或者變數可以被同一package中的其他類別，以及所有子類別訪問。
- **無修飾子（預設）**: 如果我們沒有指定存取修飾子，則該類別或成員的存取範疇被視為「package-private」。這意味著它只能被同一個package的其他類別訪問。

> 💡 `protected` 跟 `package-private` 的主要差別在於他們的存取範疇。當一個變數或方法被定義為 `protected` 時，它可以被同一個 package 中的其他類別以及所有子類別訪問，即使這些子類別不在同一個 package 中。然而，當一個變數或方法沒有明確的存取修飾子，也就是「package-private」，它只能被同一個 package 中的類別access，並不能被不同 package 的子類別access。

請注意，這些存取修飾子不僅適用於類別和成員變數，也適用於method。另外，雖然類別也可以指定存取修飾子，但是我們在實務上常常只將類別定義為public或預設（無修飾子）。

Java Collection

在Java的標準類別庫中，`java.util`套件提供了一系列的實用工具類別和集合框架。這些類別提供了許多實用的數據結構，例如列表、集合、地圖、隊列等等。以下是一些主要的 `java.util` 類別：

`java.util` 類別	描述
ArrayList	這個類別提供了一個可以動態調整大小的陣列，並且可以包含任何類型的元素。
LinkedList	提供了連結列表的數據結構，對於需要頻繁插入和刪除元素的情況來說，性能優於ArrayList。
HashSet	提供了一種不包含重複元素的集合，元素的順序不確定。
TreeSet	提供了一種不包含重複元素的集合，元素會按照自然順序或者自定義的順序進行排序。
HashMap	提供了一種基於鍵值對的數據結構，每個鍵都是唯一的，並且每個鍵對應一個值。
TreeMap	提供了一種基於鍵值對的數據結構，鍵會按照自然順序或者自定義的順序進行排序。
Stack	提供了一種後進先出（LIFO）的數據結構。
Queue	提供了一種先進先出（FIFO）的數據結構。
PriorityQueue	提供了一種基於優先級的隊列。
Date	提供了日期和時間的功能。
Random	提供了產生隨機數的功能。

這些類別使得開發者在開發Java應用程式時，可以直接使用這些預先定義好的數據結構和功能，而不需要從頭開始實作。

HashMap and ArrayList

HashMap

```
graph TD
    hashMap(HashMap) --> entry1(Entry1)
    hashMap --> entry2(Entry2)
    hashMap --> entry3(Entry3)
    entry1 --> key1(Key1)
    entry1 --> value1(Value1)
    entry2 --> key2(Key2)
    entry2 --> value2(Value2)
    entry3 --> key3(Key3)
    entry3 --> value3(Value3)
```

在這個例子中，一個`HashMap`包含了三個`Entry`物件。每一個`Entry`物件都包含了一個`Key`和一個`Value`。`HashMap`會根據`Key`的hash值來決定`Entry`的儲存位置，這使得查詢效率非常高。

在實際應用中，當我們想要快速找到一個鍵對應的值時，就可以使用`HashMap`。

舉例我們有一個`HashMap`，其中包含一些學生的ID和他們的名字，我們可以這樣來表示它：

```
HashMap<Integer, String> students = new HashMap<Integer, String>();
students.put(1, "Alice");
students.put(2, "Bob");
students.put(3, "Charlie");
```

這個`HashMap`的ASCII圖形如下所示：

```
+-------+---------+
| Key   | Value   |
+-------+---------+
| 1     | Alice   |
| 2     | Bob     |
| 3     | Charlie |
+-------+---------+
```

Key Object

HashMap在Java中被用來存(key, value)值pair。它是基於hash table的Map介面實現。以下是一個簡單的HashMap使用範例：

```
import java.util.HashMap;

public class Test {
    public static void main(String[] args) {
        // 建立一個 HashMap
        HashMap<String, Integer> prices = new HashMap<>();

        // 向 HashMap中添加元素
```

```
        prices.put("Apple", 50);
        prices.put("Orange", 20);
        prices.put("Banana", 25);

        // 輸出 HashMap
        System.out.println("HashMap: " + prices);

        // 透過鍵取得值
        int priceOfApple = prices.get("Apple");
        System.out.println("Price of Apple: " + priceOfApple);
    }
}
```

在Java的HashMap中，Key的比對是通過 `equals()` 方法和 `hashCode()` 方法來進行的。當你把key值對放入HashMap時，Java會invoke key物件的 `hashCode()` 方法來計算key的hash value，然後根據hash value決定key值對在HashMap中的存儲位置。如果兩個key物件的hash value相同，那麼Java會調用 `equals()` 方法來檢查兩個key是否真正相等。只有當 `equals()` 方法返回 `true`，兩個key才會被認為是相同的。

請注意，這也意味著如果你在自定義的類別中使用HashMap，並希望該類別的物件作為key，且在HashMap中正確地工作，你需要在該類別中覆寫 `hashCode()` 和 `equals()` 方法。未覆寫這兩個方法可能導致HashMap不能正確地存儲和檢索資料。

比如：

```
public class Key {
    private String key;

    public Key(String key) {
        this.key = key;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Key key1 = (Key) obj;
        return Objects.equals(key, key1.key);
    }

    @Override
    public int hashCode() {
        return Objects.hash(key);
    }
}
```

在此範例中，我們為 `Key` 類別覆寫了 `hashCode()` 和 `equals()` 方法，以便它的物件可以作為HashMap中的key。

ArrayList

ArrayList是一個基於array型式的資料結構，它是Java中最常用的資料結構之一。它可以動態地增加或減少元素。以下是一個簡單的ArrayList使用範例：

```
import java.util.ArrayList;

public class Test {
    public static void main(String[] args) {
        // 建立一個 ArrayList
        ArrayList<String> fruits = new ArrayList<>();

        // 向 ArrayList中添加元素
        fruits.add("Apple");
        fruits.add("Orange");
        fruits.add("Banana");

        // 輸出 ArrayList
        System.out.println("ArrayList: " + fruits);

        // 透過索引取得元素
        String firstFruit = fruits.get(0);
        System.out.println("First fruit: " + firstFruit);
    }
}
```

ArrayList的使用

Arraylist的使用，像array（可使用索引值直接取得元素，實際儲存的地方也是連續的memory space）也像list（結構大小可增減）。

```
graph LR
    node1[Node 1] --> node2[Node 2]
    node2 --> node3[Node 3]
    node3 --> node4[Node 4]
```

- **創建ArrayList**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
    }
}
```

- **新增元素**

```
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple"); // 新增元素到ArrayList的尾部
fruits.add(0, "Banana"); // 在指定的位置新增元素
```

- **刪除元素**

```
fruits.remove("Apple"); // 刪除指定元素
fruits.remove(0); // 刪除指定位置的元素
fruits.clear(); // 移除所有元素
```

- **檢索元素**

```
String fruit = fruits.get(0); // 使用索引值取得元素
int index = fruits.indexOf("Apple"); // 找到元素的索引值
boolean contains = fruits.contains("Banana"); // 檢查ArrayList是否包含某元
```

- **ArrayList的大小**

```
int size = fruits.size(); // 取得ArrayList的大小
boolean isEmpty = fruits.isEmpty(); // 檢查ArrayList是否為空
```

- **foreach ArrayList**

```
for (String fruit : fruits) { // 使用增強的for迴圈
    System.out.println(fruit);
}

fruits.forEach((fruit) -> System.out.println(fruit)); // 使用Java 8的forEa
ch方法和Lambda表達式
```

Wrapper Classes

> 📖 因為在Java Collections中能被加入/刪除/查詢的，都是object，那Primitive type的variable怎麼辦呢？我們就要使用Wrapper class

在 Java 中，每一種基本數據類型都有一個對應的包裹類別（Wrapper Class）。這些包裹類別將基本數據類型封裝為物件，使我們能夠在需要物件的情況下使用基本數據類型。例如，在集合框架中，只能儲存物件，不能直接儲存基本數據類型的值。

以下是 Java 中的基本數據類型及其對應的包裹類別：

基本數據類型	Wrapper Class
boolean	Boolean
byte	Byte

基本數據類型	Wrapper Class
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

這些包裹類別提供了一系列的方法，可以用來進行各種操作，例如轉換數據類型、比較兩個數值、進行數學運算等。

以下是一個使用Wrapper Class的範例：

```
public class Main {
    public static void main(String[] args) {
        // 建立 Integer Wrapper Class的物件
        Integer myInt = 5;
        Double myDouble = 5.99;

        // 轉換為不同的數據類型
        System.out.println(myInt.doubleValue());
        System.out.println(myDouble.intValue());

        // 比較兩個數值
        System.out.println(myInt.compareTo(3));
        System.out.println(myDouble.compareTo(5.99));

        // 進行數學運算
        System.out.println(Math.sqrt(myDouble));
    }
}
```

在這個範例中，我們首先建立了一個 `Integer` 包裹類別的物件 `myInt` 和一個 `Double` 包裹類別的物件 `myDouble`。然後，我們使用 `doubleValue()` 方法將 `myInt` 轉換為 `double` 類型，使用 `intValue()` 方法將 `myDouble` 轉換為 `int` 類型。接著，我們使用 `compareTo()` 方法來比較 `myInt` 和 `myDouble` 與其他數值的大小。最後，我們使用 `Math.sqrt()` 方法來計算 `myDouble` 的平方根。

以下是一個`HashMap`的例子，其中的`Key`是`String`類型，`Value`是`Integer`類型。其主要用途是記錄某個`String`的`Key`在資料中出現的次數：

```
import java.util.HashMap;

public class HashExample {
    public static void main(String[] args) {
        // 建立一個 HashMap
        HashMap<String, Integer> countMap = new HashMap<>();
```

```
        // 假設我們有一個字符串陣列
        String[] strings = {"apple", "orange", "banana", "apple", "orang
e", "apple"};

        // 我們遍歷這個陣列，對每個出現的字符串做計數
        for(String s: strings){
            if(countMap.containsKey(s)){
                // 如果Map中已經有這個Key，則將其值增加1
                countMap.put(s, countMap.get(s) + 1);
            } else {
                // 否則，向Map中添加這個Key，並將其值設為1
                countMap.put(s, 1);
            }
        }

        // 輸出 HashMap
        System.out.println("HashMap: " + countMap);
    }
}
```

在這個範例中，`HashMap` 的 `Key` 是 `string`，`Value` 是該 `string` 在陣列中出現的次數。這可以用來快速查詢任意 `string` 在陣列中出現的次數。

📖 在Java中，wrapper class物件可以直接使用如 `+` 運算符，這是因為Java的Auto-unboxing和Auto-boxing機制。

所以當您看到這樣的表達式 `countMap.get(s) + 1`，其實是發生了以下幾個步驟：

1. Auto-unboxing：`countMap.get(s)` 從 `HashMap` 中返回一個 `Integer` 物件。由於要進行加法運算，Java會自動將這個 `Integer` 物件unboxing成一個基本型別 `int`。
2. **加法運算**：現在，您有了一個基本型別的 `int` 值，可以直接使用 `+` 運算符將其與 `1` 相加。
3. Auto-boxing：加法運算完成後，結果是一個基本型別的 `int` 值。當您將這個值放回 `HashMap` 中時，Java會自動將這個基本型別 `int` 值boxing成一個新的 `Integer` 物件。

這個機制讓開發者能夠更加方便地處理基本型別和對應的封裝型別之間的轉換，提高了開發效率。不過，需要注意的是，過度依賴Auto-unboxing和Auto-boxing可能會導致性能問題，尤其是在大量計算的場景中，因為每一次boxing/unboxing都可能產生額外的object，從而增加GC的負擔。

❓ 聰明如你，如果你面對到的是"大量計算的場景"，那你會怎麼處理呢？以減少GC的負擔？

Auto-boxing/Auto-unboxing的使用風險

Auto-boxing和auto-unboxing是Java的一種機制，允許直接在基本型別和對應的封裝型別之間轉換。然而，這種機制可能會導致NullPointerException的風險。

當你試圖對一個null的封裝型別實例進行auto-unboxing時，Java會試圖將null轉換為一個基本型別，這時就會拋出NullPointerException。例如：

```
Integer integerObject = null;
int i = integerObject;  // 會拋出NullPointerException
```

在這個例子中，我們嘗試將一個null的Integer物件進行unboxing以轉換為int型別，Java在嘗試進行轉換時會拋出NullPointerException。

> 💡 為了避免這種問題，你需要確保在進行auto-unboxing之前，物件不是null。

在使用 Auto-boxing/auto-unboxing 機制時，使用 `==` 運算符來比較兩個 `Integer` 物件可能會產生意想不到的結果。這是因為 `==` 運算符在比較物件時，是比較物件的記憶體位址，而非其值。例如：

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b);  // 輸出 false
```

在這個例子中，即使 `a` 和 `b` 的值都是 `1000`，但由於他們是兩個不同的物件，所以他們的記憶體位址不同，因此 `a == b` 的結果是 `false`。

然而，對於介於 `-128` 至 `127` 之間的整數，Java 會自動將其緩存(caching)，所以當我們創建該範圍內的 `Integer` 物件時，Java 會直接從緩存中取得物件，而非創建新的物件。這樣一來，如果我們比較的兩個 `Integer` 物件的值都在該範圍內，使用 `==` 運算符比較的結果就會是 `true`，如下例：

```
Integer a = 100;
Integer b = 100;
System.out.println(a == b);  // 輸出 true
```

在這個例子中，由於 `a` 和 `b` 的值都在 `-128` 至 `127` 範圍內，所以他們實際上指向同一個物件，因此 `a == b` 的結果是 `true`。

> 💡 這會讓你以為使用Wrapper Class 會有跟Primitive Type數值有同樣的使用行為，但這是錯誤的喔！
> 為了避免這種風險，我們應該使用
> `equals()` 方法來比較兩個 `Integer` 物件的值，而非使用 `==` Operator。

> 🔥 既然都有了wrapper class，那為什麼java不就不要有primitive type data，都用wrapper class就好？
>
> 這是因為基本型別（Primitive Type）的效能優於封裝類別（Wrapper Class）。基本型別直接儲存值，而封裝類別則儲存一個物件，該物件包含了實際的值。因此，存取基本型別通常比存取封裝類別更快，且佔用的記憶體空間也較小。此外，基本型別的預設值為零或false，而封裝類別的預設值則是null，這在處理空值時也需要特別注意。

Iterator 介面

在 Java 中，`Iterator` 是一個介面，它是 Java Collections Framework 的成員。它用於遍歷並選取集合對象中的元素，也就是說，`Iterator` 用於逐一訪問集合中的元素。 `Iterator` 對象稱為迭代器，它設計的目的是為了提供一種方法來訪問一個聚合對象（如：陣列或集合）的元素，而又不需要暴露該對象的內部表示。

`Iterator` 介面定義了三個方法：

- `hasNext()`: 這個方法返回 `true` 如果迭代器中還有更多的元素，否則返回 `false`。
- `next()`: 這個方法返回迭代器中的下一個元素。
- `remove()`: 這個方法從集合中移除迭代器最後返回的元素。

以下是一個使用 `Iterator` 的範例程式：

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        // 建立一個 ArrayList
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        // 獲取 Iterator
        Iterator<String> it = fruits.iterator();

        // 使用 Iterator 遍歷 ArrayList
        while(it.hasNext()) {
            String fruit = it.next();
            System.out.println(fruit);

            // 如果水果是 "Banana"，則從列表中移除
            if(fruit.equals("Banana")) {
                it.remove();
            }
        }

        System.out.println("After removal:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

在這個範例中，我們首先創建了一個包含三種水果的 `ArrayList`。然後，我們使用 `iterator()` 方法獲取一個 `Iterator`。接著，我們使用 `hasNext()` 和 `next()` 方法遍歷列表。在遍歷過程中，如果遇到"Banana"，我們就使用 `remove()` 方法將其從列表中移除。

以下是如何使用Iterator來訪問HashMap中的key set和value set的範例：

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // 創建一個 HashMap
        HashMap<Integer, String> students = new HashMap<>();
        students.put(1, "Alice");
        students.put(2, "Bob");
        students.put(3, "Charlie");

        // 獲取 key set 的 Iterator
        Set<Integer> keys = students.keySet();
        Iterator<Integer> keyIterator = keys.iterator();
        while (keyIterator.hasNext()) {
            Integer key = keyIterator.next();
            System.out.println("Key: " + key);
        }

        // 獲取 value set 的 Iterator
        Collection<String> values = students.values();
        Iterator<String> valueIterator = values.iterator();
        while (valueIterator.hasNext()) {
            String value = valueIterator.next();
            System.out.println("Value: " + value);
        }
    }
}
```

在這個範例中，我們首先創建了一個HashMap `students`。然後，我們使用 `keySet()` 方法取得key的set，並使用 `iterator()` 方法獲取一個Iterator來遍歷所有的key。接著，我們使用 `values()` 方法取得value的collection，並使用 `iterator()` 方法獲取一個Iterator來遍歷所有的value。

使用 `Iterator` 的主要優點是，它提供了一種抽象的方式來遍歷集合中的元素，而不需要知道集合的內部結構。不論實際的資料結構是`ArrayList`、`HashSet`或者`TreeSet`等等，我們都可以使用相同的`Iterator`介面來存取所有的元素。換句話說，`Iterator` 提供了一種統一的遍歷方式來應對各種不同的資料結構。

Revisit Try-Catch-Final statement

在Java中，try-catch-finally語句是用於處理異常的一種結構。它讓程式在遇到異常時不會立即終止，而是可以對異常進行處理，並繼續執行下去。

- **try區塊**: 這部分包含可能會產生異常的程式碼。當這部分程式碼產生異常時，會立即跳出try區塊，並尋找對應的catch區塊進行處理。
- **catch區塊**: 這部分包含處理異常的程式碼。當try區塊中的程式碼產生異常時，會立即轉到與該異常類型相match的catch區塊進行處理。catch區塊可以有多個，用來處理不同類型的異常。
- **finally區塊**: 無論是否產生異常，finally區塊中的程式碼都會被執行。這部分一般用於清理工作，比如關閉檔案或釋放資源等。

以下是一個簡單的try-catch-finally語句的例子：

```
try {
    // 可能會產生異常的程式碼
} catch (ExceptionType1 e1) {
    // 處理 ExceptionType1 的程式碼
} catch (ExceptionType2 e2) {
    // 處理 ExceptionType2 的程式碼
} finally {
    // 一定會被執行的程式碼
}
```

在這個例子中，如果try區塊中的程式碼產生ExceptionType1的異常，則會立即跳到第一個catch區塊進行處理。如果產生ExceptionType2的異常，則會跳到第二個catch區塊進行處理。無論是否產生異常，finally區塊中的程式碼都會被執行。

當你寫一個方法並且您希望其他方法能捕獲到可能的異常時，你可以在Method使用 `throws` 關鍵字來指定可能會拋出的異常。舉例來說：

```
public void myMethod() throws ExceptionType1, ExceptionType2 {
    // 方法的程式碼
}
```

在這個例子中，`myMethod` 方法可能會拋出 `ExceptionType1` 或 `ExceptionType2`。當其他方法呼叫 `myMethod` 時，它們必須要用 try-catch 語句來處理可能的 `ExceptionType1` 和 `ExceptionType2`。

```
try {
    myMethod();
} catch (ExceptionType1 e1) {
    // 處理 ExceptionType1 的程式碼
} catch (ExceptionType2 e2) {
    // 處理 ExceptionType2 的程式碼
}
```

在這個例子中，如果 `myMethod` 拋出 `ExceptionType1`，那麼會立即跳到第一個 catch 區塊來處理。如果拋出 `ExceptionType2`，則會跳到第二個 catch 區塊來處理。

> 💡 簡單地來說，如果底層的method有丟出各種的exception而你不知道怎麼處理，你其實可以選擇throws出去。直到main() method中如果都還沒有catch來處理，也都throws出去的話，則由JVM統一將所有的error dump出來後結束程式。

以下是一個簡單，可以自行定義Exception的例子：

```
public void myMethod() throws Exception {
    // 某些條件
    if (true) {
        throw new Exception("這裡有一個異常");
    }
    // 其他程式碼
}
```

在這個例子中，`myMethod`方法在某些條件下會拋出一個新的`Exception`。這個`Exception`包含了一個描述異常的訊息。注意到，由於`myMethod`可能會拋出`Exception`，所以在Method signature我們必須使用`throws`關鍵字來指定這一點。

當其他方法呼叫`myMethod`時，它們必須使用try-catch語句來捕捉和處理這個`Exception`。

```
try {
    myMethod();
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

在這個例子中，如果`myMethod`拋出`Exception`，則會立即跳到catch區塊來處理這個異常。在catch區塊中，我們使用`getMessage`方法來獲取和輸出異常的訊息。

以下是常見的Exception：

Exception 類別	描述
NullPointerException	當應用程式嘗試在需要物件的地方使用 null，這種異常會產生
ArrayIndexOutOfBoundsException	當索引值為負或大於等於陣列大小時，這種異常會產生
ClassCastException	當嘗試將物件強制轉換為不相容的類別時，這種異常會產生
IllegalArgumentException	當傳遞非法或不適當的引數時，這種異常會產生
ArithmeticException	當出現異常的算術條件時（如除以零），這種異常會產生
IllegalStateException	在Java環境或應用程式中不適當的時間點產生的異常
NumberFormatException	當應用程式嘗試將字串轉換成一種數值型別，但該字串不能轉換為適當的格式時，這種異常會產生

Assert

在Java中，assert關鍵字用於在開發階段進行測試，用來確保程序執行時符合預期。如果assert語句的條件為真，則該語句不做任何事；如果條件為假，則會拋出一個`AssertionError`異常。

以下是assert用法的基本語法：

```
assert 表達式1 : 表達式2;
```

- 表達式1：這是一個boolean表達式，用於進行檢查。如果表達式1的結果為`true`，則整個assert語句不做任何事情。如果表達式1的結果為`false`，則會拋出一個`AssertionError`異常，並將表達式2的結果作為異常的錯誤訊息。
- 表達式2：這是一個任意類型的表達式，用於生成異常的錯誤訊息。表達式2是optional，如果省略，則錯誤訊息為`null`。

以下是一個assert的使用範例：

```
public class Main {
    public static void main(String[] args) {
        int age = 15;

        assert age >= 18 : "未成年不得入內";
        System.out.println("歡迎光臨");
    }
}
```

在這個例子中，我們使用assert語句來確保age的值大於或等於18。如果age的值小於18，則會拋出一個`AssertionError`異常，並將"未成年不得入內"作為異常的錯誤訊息。

需要注意的是，默認情況下，assert語句是不執行的。為了啟用assert語句，需要在啟動Java應用程式時加上`-ea`（enable assertions）參數。

```
java -ea Main
```

Assert 在大型軟體執行上的重要性不容忽視。它是一種程式設計工具，可以用來進行偵錯和測試，確保程式在運行時符合預期的條件。當程式的某個特定條件為假時，Assert 會拋出 AssertionError，這有助於開發者在早期階段就發現並修正錯誤，而不是等到軟體已經部署並運行一段時間後才發現問題。

在大型軟體開發過程中，由於程式的複雜性和規模，要確保所有的功能都能正確運行是一項極具挑戰的任務。在這種情況下，Assert 的作用就顯得尤為重要，它可以幫助開發者確保程式的關鍵部分運行正確，並且在出現問題時能夠迅速地找到錯誤的源頭。

通常，我們會在以下幾種情況下使用 Assert：

- 驗證程式的內部狀態：如果程式的某個內部狀態不符合預期，可以使用 Assert 來檢查並拋出錯誤。這樣可以提早發現問題，防止錯誤的內部狀態導致更嚴重的問題。
- 驗證輸入參數：如果一個方法對輸入參數有特定的要求，可以使用 Assert 來驗證輸入參數是否符合這些要求。這可以防止無效的輸入引發錯誤。
- 在測試中驗證預期結果：在寫單元測試或整合測試的時候，我們經常使用 Assert 來驗證方法的返回值是否與預期結果相符。

總的來說，Assert 是一種有效的程式設計工具，對於提高程式的穩定性和可靠性有著重要的作用。

`assert`運在用如c++等machine-code optimization完，在deploy之後，可能難以trace bug的情況(通常只會丟出`segv`及dump出一些難以理解的machine message)，將能十分有效的協助在客戶端印出必要的訊息。

`assert`在java的功能就沒有c++等語言那麼明顯，因為JRE能將所有的stack trace都記錄下來，一發生問題，通常容易初步反向追查。但在不明顯的bug上，`assert`的效用一樣很大。

環境變數讀取

在 Java 中，我們可以利用 `System.getenv()` 方法來讀取作業系統的環境變數。這個方法會返回一個 `Map`，其中的 key 是環境變數的名稱，value 是環境變數的值。以下是一個簡單的範例：

```
public class Main {
    public static void main(String[] args) {
        // 獲取所有的環境變數
        Map<String, String> env = System.getenv();

        // 印出每一個環境變數
        for (String envName : env.keySet()) {
            System.out.format("%s=%s\n", envName, env.get(envName));
        }
    }
}
```

在這個例子中，我們首先呼叫 `System.getenv()` 來獲取所有的環境變數，然後逐一印出每一個環境變數的名稱和值。

如果我們只需要讀取特定的環境變數，我們可以直接傳入環境變數名稱作為參數給 `System.getenv()` 方法，如下例：

```
public class Main {
    public static void main(String[] args) {
        // 讀取 "PATH" 環境變數
        String path = System.getenv("PATH");

        // 印出 "PATH" 環境變數的值
        System.out.println("PATH = " + path);
    }
}
```

在這個例子中，我們讀取了 "PATH" 環境變數，並印出了它的值。

📖 在 Linux 系統下，我們可以使用 `export` 命令來設定環境變數。以下是一個範例：

```
export VAR_NAME="value"
```

在這個例子中，我們創建了一個名為 `VAR_NAME` 的環境變數，並將其值設為 `value`。

透過assert跟environment value來進行debug技巧

以下是一個Java程式碼的範例，該程式碼將assert和System.getenv()指令結合使用，作為一種有效的debug策略：

```
public class Main {
    public static void main(String[] args) {
        // 讀取 "DEBUG" 環境變數
        String debug = System.getenv("DEBUG");

        // 將 "DEBUG" 環境變數轉換為 boolean 值
        boolean isDebug = Boolean.parseBoolean(debug);

        // 如果 DEBUG 環境變數被設定為 "true"，則啟用 assert
        if (isDebug) {
            // 這裡是一個示例，檢查某個條件是否為真
            int x = 10;
            assert x == 10 : "x 應該為 10";
        }
    }
}
```

在這個例子中，我們首先從環境變數讀取 "DEBUG"，並將其轉換為boolean值。如果DEBUG環境變數被設定為 "true"，則我們會啟用assert來檢查某個條件是否為真。

這種策略的好處是，我們可以在debug期間啟用assert，並在客戶環境中關掉它。這可以讓我們在開發和測試期間更容易找到和修復問題，同時在客戶線上環境中避免不必要的time overhead。