

Inheritance & polymorphism

繼承 & 多型

Java Fundamental



Content

- ◆ 繼承(Inheritance)
- ◆ 多型(Polymorphism)

Content

◆ 繼承(Inheritance)

- 繼承觀念
- 繼承實作
- 方法覆寫
- 繼承關係下的物件建構

◆ 多型(Polymorphism)

繼承

- ◆ 繼承是類別間之關係，在此關係中某類別之資料結構與行為可供其關係中之類別分享。
- ◆ 繼承者稱為子類別(Subclass)，被繼承者稱為父類別(Superclass)。
- ◆ Java使用extends來表示繼承的關係。

- ◆ 父類別定義的屬性與方法，子類別當然也適用，不過要成為子類別。

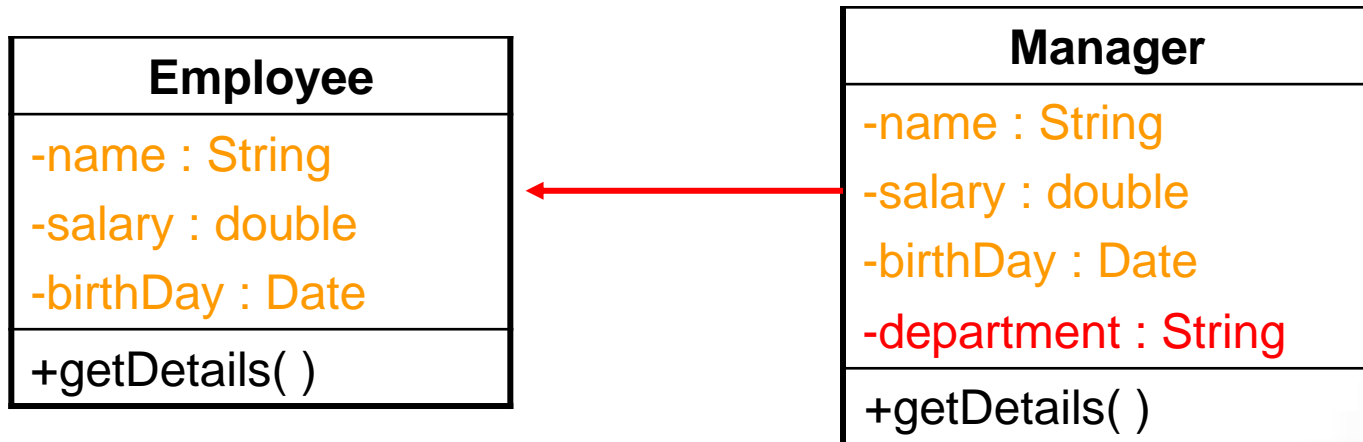
必須透過繼承(使用extends關鍵字)。當然子類別也可以自行定義新的成員，以表現本身的特質，但是父類別物件無法存取子類別定義的成員。

- ◆ final類別不可被繼承

- final class Book{}
- class ComputerBook extends Book{} //編譯失敗，因為Book類別不可被繼承

繼承

- ◆ 繼承讓類別的程式碼可以延伸及重複使用
 - 父類別 base class/superclass
 - 子類別 derived class/subclass
 - 類別延伸(extends class)的關係是 "is a" 的概念。



A Manager is a Employee

繼承

- ◆ 下例中，類別 D 繼承類別 C：

```
class C {
```

```
...
```

```
}
```

```
class D extends C {
```

```
...
```

```
}
```

- ◆ 當類別 D 繼承了類別 C 後，類別 C 中一切可以被繼承的事物 (包括所有 非 private 的資源) 都將變成類別 D 中的一部分。

繼承範例

```
1. class C {
2.     private int i;
3.     public void setInfo(int x) { i = x; }
4.     public int getInfo() { return i; }
5. }
6. class D extends C {} // 類別 D 繼承類別 C
7. class E extends C {} // 類別 E 繼承類別 C

8. public class F {
9.     public static void main(String[] args) {
10.         D d = new D();
11.         E e = new E();

12.         d.setInfo(5);
13.         e.setInfo(7);

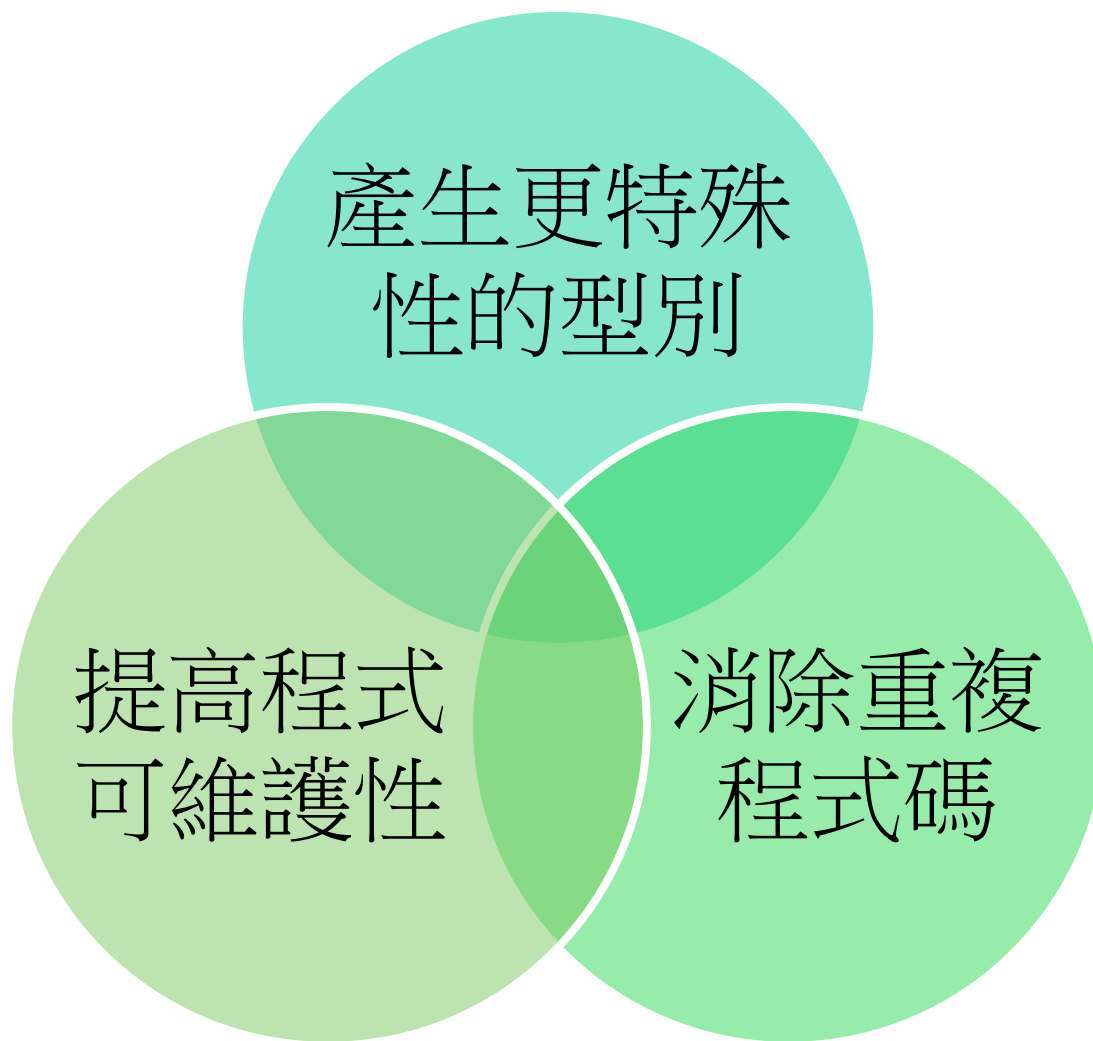
14.         System.out.println("The value of d is "+d.getInfo());
15.         System.out.println("The value of e is "+e.getInfo());
16.     }
17. }
```

執行結果：

The value of d is 5

The value of e is 7

繼承機制的優點



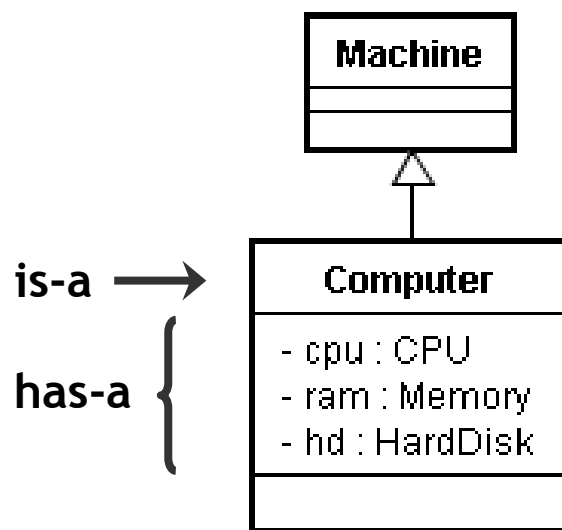
is-a 與 has-a

◆ is a (是一個) :

- 延伸的關係。
- Java 語言利用 **extends** 關鍵字來實作
- EX : 電腦是一種電子機械產品
 - 電腦類別繼承了電子機械產品類別。
 - 電子機械產品是父類別，而電腦則是子類別。

◆ has a (有一個) :

- 聚合的關係。
- 類別中的成員變數(member variable)來表示。
- EX : 電腦中有 CPU、256M RAM、40GB HD
 - CPU、RAM 與 HD 便成了電腦的成員變數。



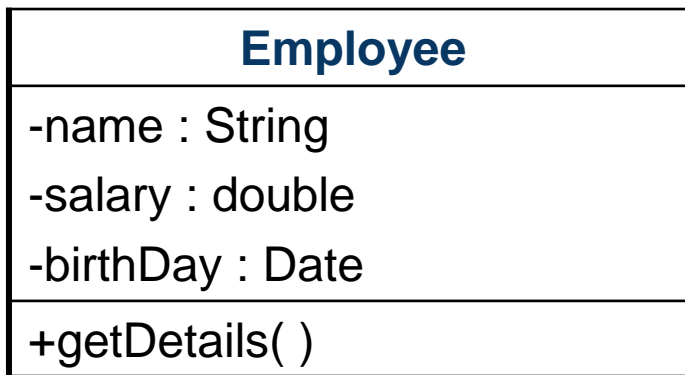
Java 技術實作繼承

◆ 建立類別語法

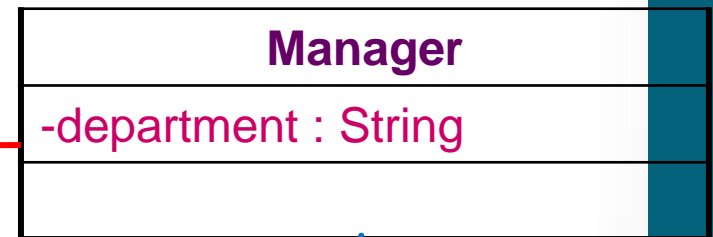
[modifier] class 子類別名稱 extends 父類別名稱 {

//類別內容

}



Manager 繼承了 Employee 所有成員;
屬性:name, salary, birthDay
方法:getDetails()方法



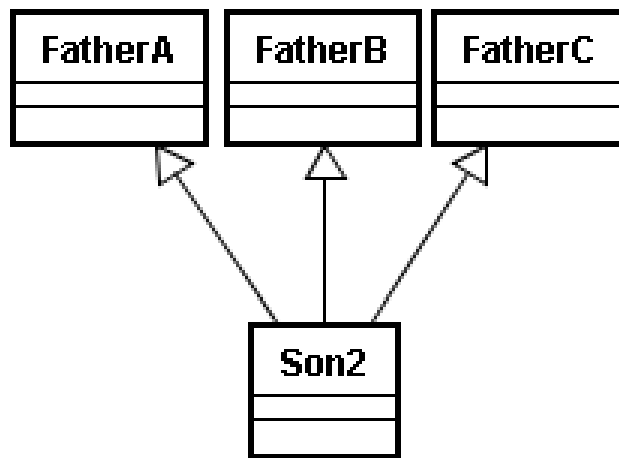
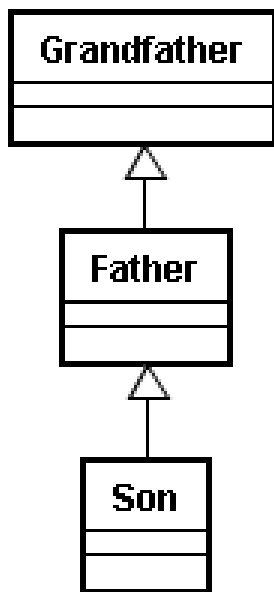
```
01 public class Employee {
02     private String name = "Sean";
03     private double salary = 10000;
04     public void getDetails( ) {
05         System.out.println("Name:" + name);
06         System.out.println("Salary:" + salary);
07     }
08 }
```

```
01 public class Manager extends Employee {
02     private String dept = "EDU";
03
04 }
05
```

單一繼承

- ◆ Java 語言在繼承上只允許單一繼承(Single Inheritance)關係。
- ◆ 子類別在定義繼承的關係時，只能針對單一父類別做延伸，不能同時使用來自多個父類別的資源。

單一繼承(Java支援)



多重繼承(C++等...支援)

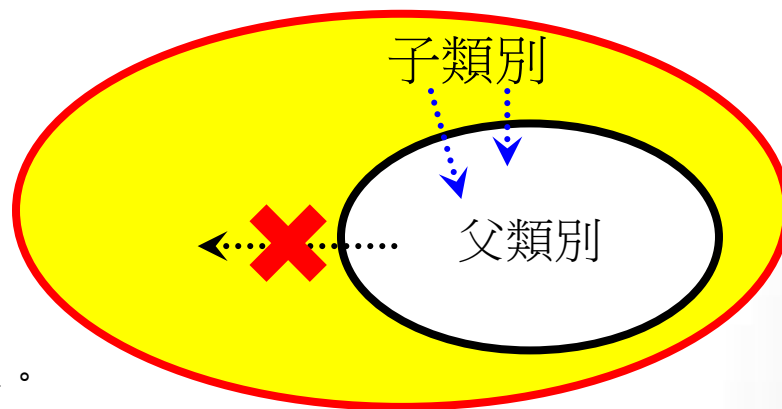
繼承中的資源使用

◆ 當子類別繼承父類別，擁有父類別中的資源

- 子類別繼承(擁有)父類別成員(屬性和方法)
 - 實作：子類別中包含父類別
- 父類別不擁有子類別屬性和方法
- **建構子不會被繼承**

◆ 子類別存取父類別資源

- 存取父類別的屬性：直接使用屬性名稱。
- 呼叫父類別的方法：直接使用方法名稱(參數列)。
- 但存取權限仍受限於父類別的存取修飾字。



繼承範例

```
01 class Father {  
02     public int money = 1000000; ←  
03     public void undertaking() { ←  
04         System.out.println("父親的事業");  
05     }  
06 }
```

```
01 class Son extends Father{  
02  
03 }
```

```
01 public class Extends {  
02     public static void main(String[] args) {  
03         Son son = new Son();  
04         son.undertaking(); ←  
05         System.out.println("金額:" + son.money); ←  
06     }  
07 }
```



```
C:\JavaClass>javac Extends.java  
C:\JavaClass>java Extends  
父親的事業  
金額:1000000  
C:\JavaClass>
```

覆寫(overriding)

◆ 子類別將由父類別繼承來的屬性(變數、資料結構)或方法(行為)重新定義的動作稱為覆寫(overriding)。

◆ 方法覆寫

➤ 將繼承自父類別的方法遮蓋起來，使得以子類別所產生的物件不能再使用已經被覆蓋的方法。

➤ 這種設計在繼承類別時，可以將不符合需求的方法加以改寫，如此一來，子類別的使用者就不會也不能看到及使用父類別中的方法，達到重新設計的目的。

◆ 變數覆寫

➤ 將繼承自父類別的變數遮蓋起來。事實上，當變數產生覆蓋問題時，在父類別中的變數宣告並不會因此而消失，它只是隱藏起來而已。

變數覆寫範例

```
1. class C {  
2.     int i = 10;  
3. }  
4. class D extends C {  
5.     int i = 5;  
6. }  
7. public class E {  
8.     public static void main(String[] args) {  
9.         D d = new D();  
10.        System.out.println("d = "+d.i);  
11.    }  
12.}
```

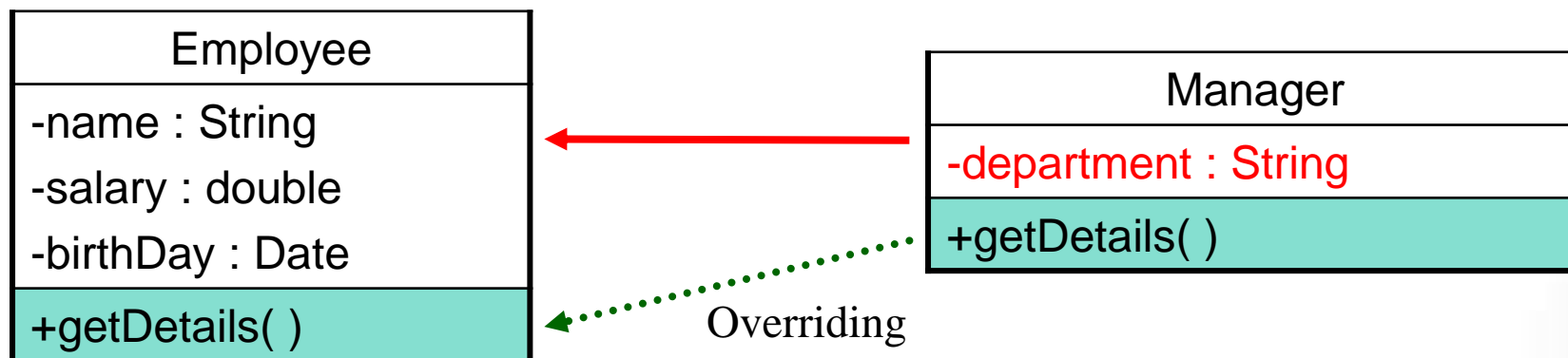
執行結果：

d = 5

方法覆寫 Method Override

◆ 方法覆寫 Method Override

- 子類別改寫父類別中相同的名稱及參數列的方法



方法覆寫 Method Override

```
01 public class Employee {  
02     private String name = "Sean";  
03     private double salary = 10000;  
04     public void getDetails() {  
05         System.out.println("Name:" + name);  
06         System.out.println("Salary:" + salary);  
07     }  
08 }
```

```
01 public class Test {  
02     public static void main(String [ ] args) {  
03         Employee e = new Employee();  
04         e.getDetails();  
05         Manager m = new Manager ();  
06         m.getDetails();  
07     }  
08 }
```

```
01 public class Manager extends Employee {  
02     private String dept = "EDU";  
03     @Override  
04     public void getDetails() {  
05         System.out.println("Name:" + name);  
06         System.out.println("Salary:" + salary);  
07         System.out.println("Department:"+dept);  
08     }  
09 }
```

呼叫被覆寫的方法 - super

◆ super 關鍵字

➤ 子類別物件中欲參考父類別物件的屬性、方法及建構子。

- super.屬性
- super.方法(參數列)
- super(參數列)

➤ super關鍵字必須在繼承關係的運作下才有意義。

```
public class Employee {  
    private String name = "Sean";  
    private double salary = 10000;  
    public void getDetails() {  
        System.out.println("Name:" + name);  
        System.out.println("Salary:" + salary);  
    }  
}
```

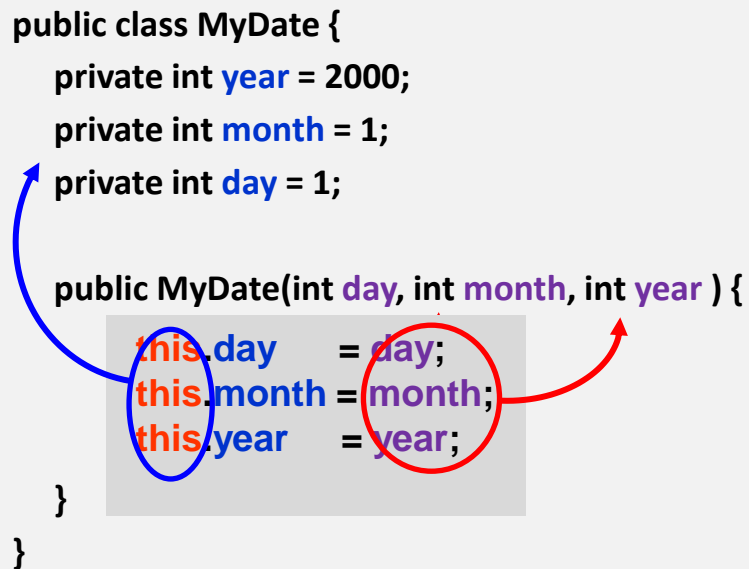
```
public class Manager extends Employee {  
    private String dept = "EDU";  
    public void getDetails() {  
        super.getDetails();  
        System.out.println("Department:"+dept);  
    }  
}
```

存取被遮蔽的屬性 - this

◆ this 關鍵字

- 編譯時期自動加入,代表本身物件的參考(Reference)
- 方法或建構子中,欲參考被遮蔽的物件屬性及方法
 - this.屬性
 - this.方法(參數列)

```
public class MyDate {  
    private int year = 2000;  
    private int month = 1;  
    private int day = 1;  
  
    public MyDate(int day, int month, int year) {  
        this.day = day;  
        this.month = month;  
        this.year = year;  
    }  
}
```



Default Constructor

```
public class Employee {  
    private String name = "Sean";  
    private double salary = 10000;  
    public Employee() {  
        super();  
    }  
    public void getDetail() {  
        System.out.println("Name:" + name);  
        System.out.println("Salary:" + salary);  
    }  
}
```

Manager m = new Manager();

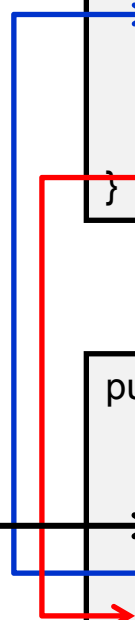
```
public class Manager extends Employee {  
    private String deptName = "EDU";  
    public Manager() {  
        super();  
    }  
    public void getDetail() {  
        super.getDetail();  
        System.out.println("Department:" + dept);  
    }  
}
```

Constructors & super()

```
public class Employee {  
    private String name = "Sean";  
    private double salary = 10000;  
  
    public Employee(String n, double s) {  
        name = n;  
        salary = s;  
    }  
}
```

```
public class Manager extends Employee {  
    private String dept = "EDU";  
  
    public Manager(String n, double s, String d) {  
        super(n, s);  
        dept = d;  
    }  
}
```

Manager m = new Manager
("Sean", 50000.0, "EDU");



Inheritance & Constructor

- ◆ 若使用**super**或**this**呼叫其他建構式，該呼叫式必須放在建構式區塊內的第1行

```
ComputerBook(String n, double p, String a, boolean h){  
    super(n, p, a); //正確，放在建構式內的第1行  
    hasDisk = h;  
}
```

```
ComputerBook(String n, double p, String a, boolean h){  
    hasDisk = h;  
    super(n, p, a); //錯誤，必須放在建構式內的第1行  
}
```

◆ 建構式進階觀念

- 任何類別的建構式內都必須存在呼叫父類別建構式的呼叫式，如果沒有則編譯器會自動加上呼叫父類別預設建構式的呼叫式 - 「**super();**」。
- 如果類別內沒有任何建構式，則會加上1個含有「**super();**」的預設建構式。
如果該類別沒有父類別，則「**super();**」代表**呼叫Object類別的預設建構式**。

Constructors overloading & this()

MyDate d = new MyDate (27, 6, 2011);

```
public class MyDate {  
    private int year = 2000;  
    private int month = 1;  
    private int day = 1;  
  
    public MyDate(int d, int m, int y ) {  
        super();  
        year = y;  
        month = m;  
        day = d;  
    }  
  
    public MyDate(int d, int m) {  
        this(d, m, 2013);  
    }  
  
    public MyDate(int d) {  
        this(d, 5);  
    }  
}
```

MyDate d = new MyDate (27);

Content

- ◆ 繼承(Inheritance)

- ◆ 多型(Polymorphism)

- 多型的特性

- 型別轉型

多型 Polymorphism

- ◆ 同名異式，簡稱為多型。
 - 一個方法可以有許多型式，也就是**相同的方法名稱**，定義**不同的實作**(implementation)。
 - 目的是希望簡化系統發展的複雜性並增加其彈性。
- ◆ **多型**在程式執行時，呼叫方法是以**動態連結(Dynamic Binding)**方式，判斷當時被呼叫物件所屬的類別來決定執行那一實作，所以又稱為動態多型。
- ◆ 動態多型是**建立在繼承的架構上**。

多型應用 Polymorphism

```
1. abstract class Shape {
2.     public abstract void f();
3. }
4. class Triangle extends Shape {
5.     public void f() {
6.         System.out.println("Triangle!");
7.     }
8. }

9. class Rectangle extends Shape {
10.    public void f() {
11.        System.out.println("Rectangle!");
12.    }
13.}

14. class Circle extends Shape {
15.    public void f() {
16.        System.out.println("Circle!");
17.    }
18.}
```

```
19. public class E {
20.     public static void main (String[] args) {
21.         Shape[] s = new Shape[] { new Triangle(),
22.                                     new Rectangle(), new Circle() };
23.
24.         for (int i=0; i<s.length; i++) {
25.             s[i].f();
26.         }
27.     }
28. }
```

執行結果：

Triangle!

Rectangle!

Circle!

多型 Polymorphism

- ◆ 當呼叫方法時，依參數的數目與類型來決定執行那個實作，此稱為多重定義、過荷或超載(Overloading)。
- ◆ 超載又稱為靜態多型。
- ◆ 因此，在設計方法過荷時需要特別小心，千萬不能有任何方法的參數個數及型態是完全一致的。

多型 Polymorphism

- ◆ `int add(int i, int j) { return i + j; }`
- ◆ `float add(float i, float j) { return i + j; }`
- ◆ `double add(double i, double j) { return i + j; }`
- ◆ `int add(int i, float f) { return i + (int) f; }`
- ◆ `int add(int i, int j, int k) { return i + j + k; }`

- ◆ `add(1, 2)`
- ◆ `add(1, 2, 3)`

多型 Polymorphism

◆ 多型的意義

- 一個物件可以用多種形態來看待。
- 型態間需有繼承關係：子類別可以被看待為父類別。

◆ Java技術實作多型

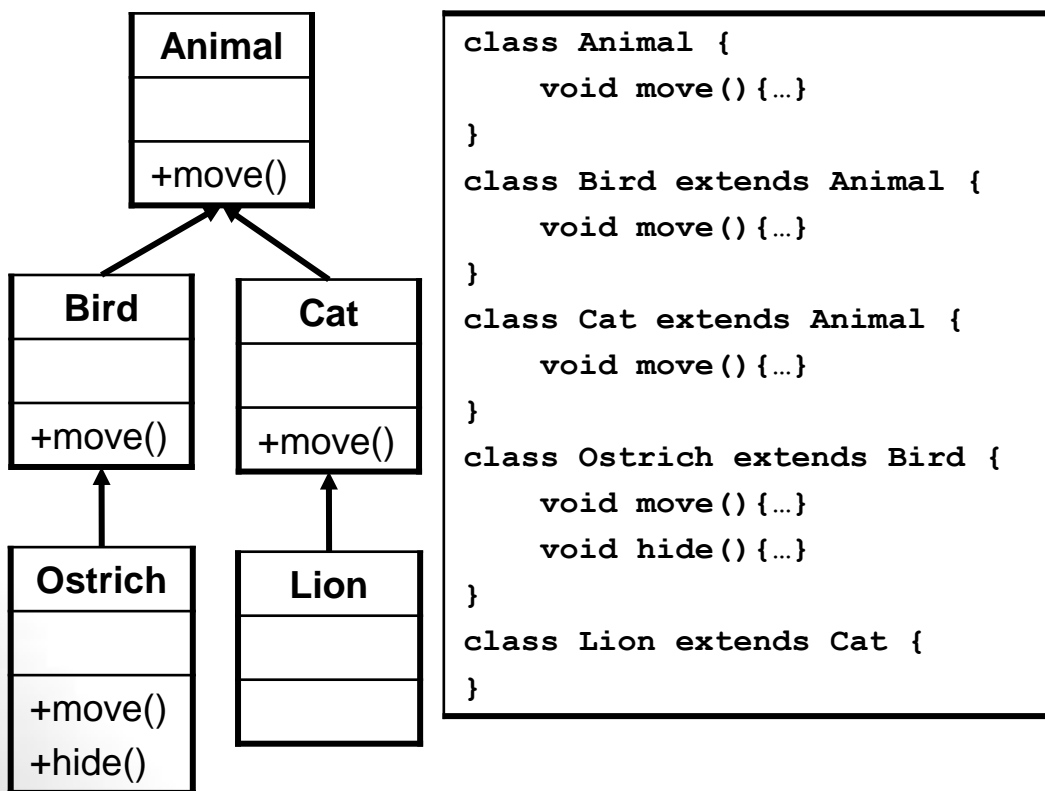
- 具有繼承關係的架構下,物件實體可以被視為多種型別。
- 將子類別物件參考指定給父類別變數。

父類別 變數名稱 = new **子類別建構子()**;

物件多型

◆ 獅子是貓科動物，所有貓科動物皆是動物。

◆ 鴕鳥是鳥類，所有鳥類皆是一種動物。



`Lion l = new Lion();`
用 Lion 獅子的眼光來看 Lion

`Cat c = new Lion();`
用 Cat 貓科動物的眼光來看 Lion

`Animal a = new Lion();`
用 Animal 動物的眼光來看 Lion

~~`Lion l1 = new Cat();`~~
用 Lion 獅子的眼光來看所有的 Cat 貓科動物

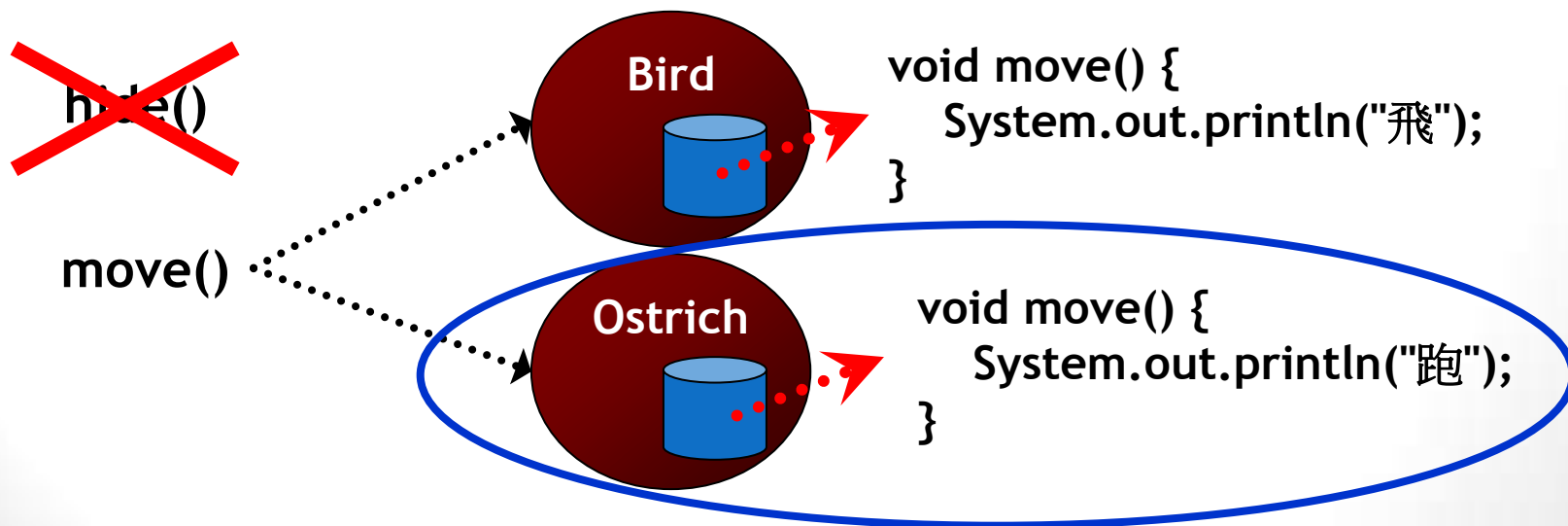
編譯錯誤

多型的特性

◆多型的特性

- 不同型態表示並不會改變原來的實體。
- 將物件視為父類別,只能用父類別有定義之屬性及方法。
- 若父類別方法被子類別覆寫,多型時,用父類別的觀點呼叫,仍會執行子類別的方法。

```
Bird bird = new Ostrich();
```



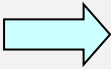
多型範例

```
class Animal {  
    void move() {  
        System.out.println("動");  
    }  
}
```

```
class Bird extends Animal {  
    void move() {  
        System.out.println("飛");  
    }  
}
```

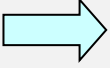
```
class Ostrich extends Bird {  
    void move() {  
        System.out.println("跑");  
    }  
    void hide() {  
        System.out.println("頭埋在土裡");  
    }  
}
```

```
Ostrich ostrich = new Ostrich();  
ostrich.move();  
ostrich.hide();
```



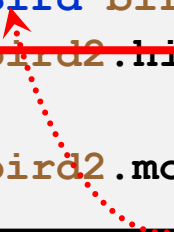
跑
頭埋在土裡

```
Bird bird1 = new Bird();  
bird1.move();
```

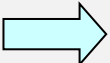


飛


```
Bird bird2 = new Ostrich();  
bird2.hide();
```



```
bird2.move();
```



跑

在 **Bird** 型別中並不知道有 **hide()** 方法

型別檢查與虛擬方法調用

◆ Java型別檢查

- 編譯時期：compiler會以**宣告的型別**作型別檢查

確保物件被視為父類別, 只能用父類別定義之屬性及方法

- 執行時期：JVM會以**實際的型別**作型別檢查

確保多型時, 用父類別的觀點呼叫, 仍會執行子類別的方法

- 虛擬方法使用(呼叫) Virtual Method Invocation

Java程式會使用(呼叫)變數在執行時期所參考之物件的行為,而不是在編譯時期

宣告類別的行為

多型範例

```
public class Employee {  
    private String name = "Sean";  
    private double salary = 10000;  
    public void getDetails() {  
        System.out.println("Name:" + name);  
        System.out.println("Salary:" + salary);  
    }  
}
```

```
public class Manager extends Employee {  
    private String dept = "EDU";  
    public void getDetails() {  
        super.getDetails();  
        System.out.println("Department:" + dept);  
    }  
    public void getDepartment() {  
        System.out.println("Department:" + dept);  
    }  
}
```

```
public class Test {  
    public static void main(String [] args) {  
        Employee e = new Employee();  
        e.getDetails();  
        Manager m = new Manager();  
        m.getDetails();  
  
        Employee p = new Manager();  
  
        p.getDetails();  
        p.getDepartment();  
    }  
}
```



Employee type
Manager instance

多型範例

```
public class Employee {  
    private String name = "Sean";  
    private double salary = 10000;  
    public void getDetails() {  
        System.out.println("Name:" + name);  
        System.out.println("Salary:" + salary);  
    }  
}
```

```
public class Manager extends Employee {  
    private String dept = "EDU";  
    public void getDetails() {  
        super.getDetails();  
        System.out.println("Department:" + dept);  
    }  
    public void getDepartment() {  
        System.out.println("Department:" + dept);  
    }  
}
```

```
public class Test {  
    public static void main(String [] args) {  
        Employee e = new Employee();  
        e.getDetails();  
        Manager m = new Manager();  
        m.getDetails();  
  
        Employee p = new Manager();  
  
        {  
            p.getDetails();  
            p.getDepartment();  
        }  
    }  
}
```

Compile-Time Type
Employee

多型範例

```
public class Employee {  
    private String name = "Sean";  
    private double salary = 10000;  
    public void getDetails() {  
        System.out.println("Name:" + name);  
        System.out.println("Salary:" + salary);  
    }  
}
```

```
public class Manager extends Employee {  
    private String dept = "EDU";  
    public void getDetails() {  
        super.getDetails();  
        System.out.println("Department:" + dept);  
    }  
    public void getDepartment() {  
        System.out.println("Department:" + dept);  
    }  
}
```

```
public class Test {  
    public static void main(String [] args) {  
        Employee e = new Employee();  
        e.getDetails();  
        Manager m = new Manager();  
        m.getDetails();  
  
        Employee p = new Manager();  
  
        p.getDetails();  
        // p.getDepartment();  
    }  
}
```

Run-Time Type
Manager



```
graph TD
    RTM[Run-Time Type Manager] --> ManagerClass[Manager Class]
    RTM --> pGetDetails[p.getDetails() call]
```

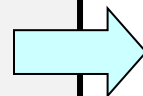
強制轉型

◆ 強制轉型

- 將被宣告為父類別的子類別物件轉型回子類別。
- (目標類別名稱)物件名稱。
- 可先用 <物件名稱> instanceof <類別名稱> 檢查。

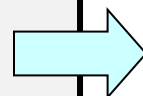
◆ 藉由轉型來解決呼叫hide()方法的問題。

```
Bird bird1 = new Bird();  
bird1.move();
```



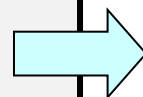
飛

```
Bird bird2 = new Ostrich();  
bird2.move();
```



跑

```
((Ostrich) bird2).hide();
```



頭埋在土裡

Instanceof 運算子

◆ instanceof 運算子

- 確認物件是否為某種類別型態
- <物件名稱> instanceof <類別名稱>
- 回傳布林值
 - **true**：該變數所參考的物件可以轉換成特定類別。
 - **false**：反之則否。

```
Bird bird1 = new Bird();  
bird1.move();
```

```
Bird bird2 = new Ostrich();  
bird2.move();
```

```
if (bird2 instanceof Ostrich) {  
    ((Ostrich) bird2).hide();  
}
```

飛

跑

頭埋在土裡

Q & A