

# Java程式設計進階

## Java 介面進階設計

鄭安翔

ansel\_cheng@hotmail.com

# 課程大綱

## 1) 介面進階

- ❑ **default** 方法
- ❑ **static** 方法

## 2) Design pattern

- ❑ DAO 設計模式
- ❑ Factory 設計模式

# 用 interface 解決問題

- 用 interface 解決問題
  - 一家五金公司販售差異性極高的幾種商品
    - 石頭 (Rocks, 以磅計重)
    - 油漆 (Paint, 以加侖計算容積)
    - 零件 (Widgets, 以個數計算)
  - 希望在財務報表上有一致的呈現方式
    - 售價 (Sales Price)
    - 成本 (Cost)
    - 利潤 (Profit)

# 範例 – 用interface 解決問題

Rock
<ul style="list-style-type: none"><li>- name:String</li><li>- unitPrice:double</li><li>- unitCost:double</li><li>- weight:double</li></ul>
+ Rock(p:double, c:double, w: double)

Paint
<ul style="list-style-type: none"><li>- name:String</li><li>- unitPrice:double</li><li>- unitCost:double</li><li>- volume:double</li></ul>
+ Paint(p:double, c:double, v: double)

Widget
<ul style="list-style-type: none"><li>- name:String</li><li>- unitPrice:double</li><li>- unitCost:double</li><li>- quantity:int</li></ul>
+ Paint(p:double, c:double, q: int)

```
public class Rock {  
    private String name = "Rock";  
    private double unitPrice;  
    private double unitCost;  
    private double weight;  
  
    public Rock(double p, double c, double w) {  
        unitPrice = p;  
        unitCost = c;  
        weight = w;  
    }  
}
```

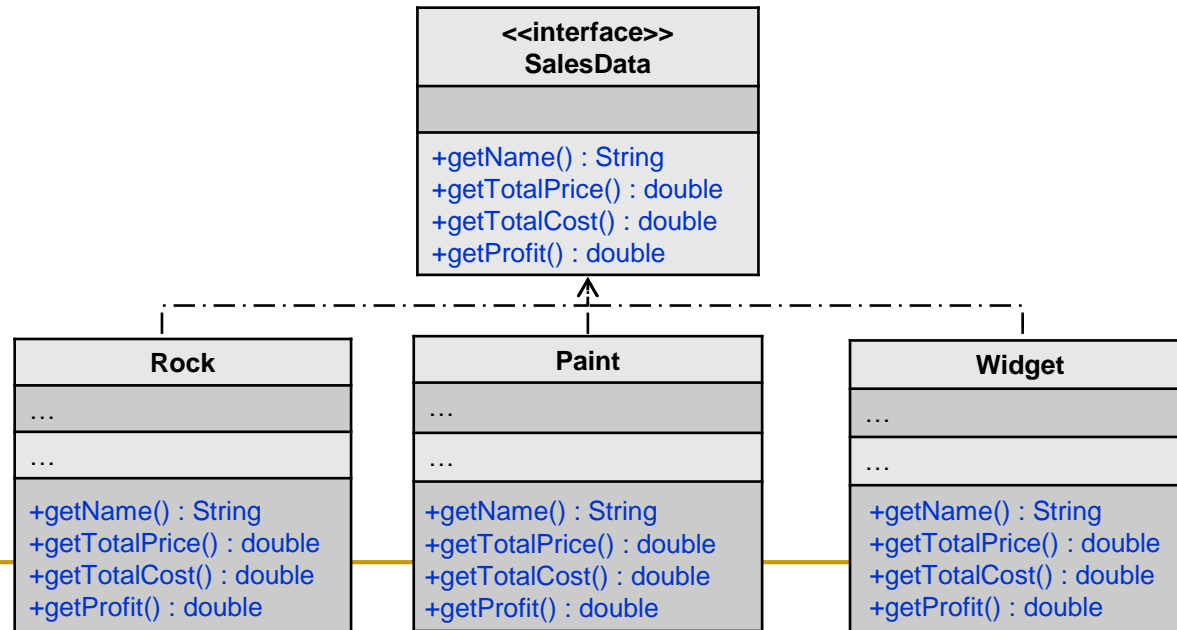
```
public class Paint {  
    private String name = "Paint";  
    private double unitPrice;  
    private double unitCost;  
    private double volume;  
  
    public Paint(double p, double c, double v) {  
        unitPrice = p;  
        unitCost = c;  
        volume = v;  
    }  
}
```

```
public class Widget {  
    private String name = "Widget";  
    private double unitPrice;  
    private double unitCost;  
    private int quantity;  
  
    public Widget(double p, double c, int q) {  
        unitPrice = p;  
        unitCost = c;  
        quantity = q;  
    }  
}
```

# 範例 – 用interface 解決問題

## ■ SalesData interface

- ❑ 定義在財務表報上需要出現的資訊
- ❑ 所有商品均需要實作SalesData介面以提供資訊



# 範例 – 用interface 解決問題

```
public interface SalesData {  
    public String getName();  
    public double getTotalPrice();  
    public double getTotalCost();  
    public double getProfit();  
}
```

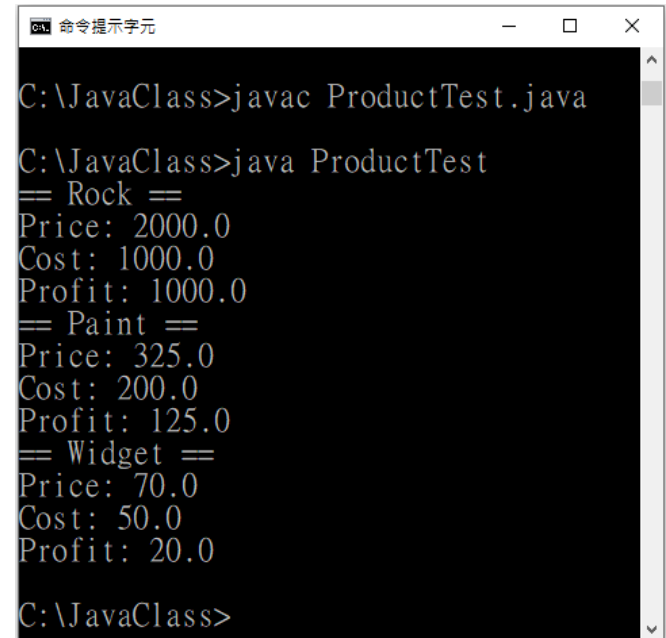
```
public class Rock implements SalesData {  
    ....  
    public String getName() {  
        return name;  
    }  
    public double getTotalPrice() {  
        return unitPrice*weight;  
    }  
    public double getTotalCost() {  
        return unitCost*weight;  
    }  
    public double getProfit() {  
        return (unitPrice-unitCost)*weight;  
    }  
}
```

```
public class Paint implements SalesData {  
    ....  
    public String getName() {  
        return name;  
    }  
    public double getTotalPrice() {  
        return unitPrice*volume;  
    }  
    public double getTotalCost() {  
        return unitCost*volume;  
    }  
    public double getProfit() {  
        return (unitPrice-unitCost)*volume;  
    }  
}
```

```
public class Widget implements SalesData {  
    ....  
    public String getName() {  
        return name;  
    }  
    public double getTotalPrice() {  
        return unitPrice*quantity;  
    }  
    public double getTotalCost() {  
        return unitCost*quantity;  
    }  
    public double getProfit() {  
        return (unitPrice-unitCost)*quantity;  
    }  
}
```

# 範例 – 用interface 解決問題

```
public class ProductTest {  
    public static void main(String args[]) {  
        Rock rock = new Rock(20, 10, 100);  
        Paint paint = new Paint(13.0, 8.0, 25.0);  
        Widget widget = new Widget(7.0, 5.0, 10);  
  
        System.out.println("== " + rock.getName() + " ==");  
        System.out.println("Price: " + rock.getTotalPrice());  
        System.out.println("Cost: " + rock.getTotalCost());  
        System.out.println("Profit: " + rock.getProfit());  
  
        System.out.println("== " + paint.getName() + " ==");  
        System.out.println("Price: " + paint.getTotalPrice());  
        System.out.println("Cost: " + paint.getTotalCost());  
        System.out.println("Profit: " + paint.getProfit());  
  
        System.out.println("== " + widget.getName() + " ==");  
        System.out.println("Price: " + widget.getTotalPrice());  
        System.out.println("Cost: " + widget.getTotalCost());  
        System.out.println("Profit: " + widget.getProfit());  
    }  
}
```



```
C:\JavaClass>javac ProductTest.java  
  
C:\JavaClass>java ProductTest  
== Rock ==  
Price: 2000.0  
Cost: 1000.0  
Profit: 1000.0  
== Paint ==  
Price: 325.0  
Cost: 200.0  
Profit: 125.0  
== Widget ==  
Price: 70.0  
Cost: 50.0  
Profit: 20.0  
  
C:\JavaClass>
```

# 工具類別提供一致的呈現方式

- 工具類別utility class :
  - 產出商品報表
    - 將多個操作整合成一個標準顯示格式
  - show()傳入SalesData介面的實作

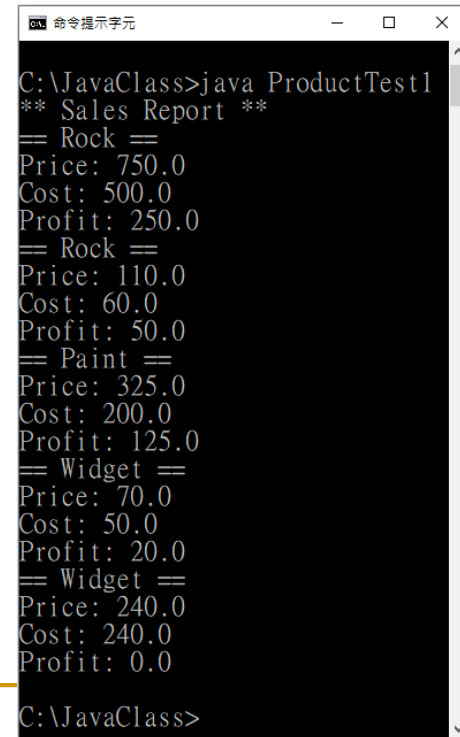
```
public class SalesTool {  
    public void show (SalesData item) {  
        System.out.println("== " + item.getName() + " ==");  
        System.out.println("Price: " + item.getTotalPrice());  
        System.out.println("Cost: " + item.getTotalCost());  
        System.out.println("Profit: " + item.getProfit());  
    }  
}
```



# 工具類別提供一致的呈現方式

- 各式商品均實作 **SalesData** 介面
- 使用 **SalesData** 作為各式商品的參考型別

```
public class ProductTest1 {  
    public static void main(String args[]) {  
        SalesData[] itemList = new SalesData[5];  
        SalesTool tool = new SalesTool();  
        itemList[0] = new Rock(15.0, 10.0, 50.0);  
        itemList[1] = new Rock(11.0, 6.0, 10.0);  
        itemList[2] = new Paint(13.0, 8.0, 25.0);  
        itemList[3] = new Widget(7.0, 5.0, 10);  
        itemList[4] = new Widget(12.0, 12.0, 20);  
        System.out.println("** Sales Report **");  
        for (SalesData item : itemList) {  
            tool.show(item);  
        }  
    }  
}
```



```
C:\JavaClass>java ProductTest1  
** Sales Report **  
== Rock ==  
Price: 750.0  
Cost: 500.0  
Profit: 250.0  
== Rock ==  
Price: 110.0  
Cost: 60.0  
Profit: 50.0  
== Paint ==  
Price: 325.0  
Cost: 200.0  
Profit: 125.0  
== Widget ==  
Price: 70.0  
Cost: 50.0  
Profit: 20.0  
== Widget ==  
Price: 240.0  
Cost: 240.0  
Profit: 0.0  
C:\JavaClass>
```

# 工具類別提供一致的呈現方式

- SalesTool 工具類別
  - 只有一個方法
  - 只用來處理實作 SalesData 的商品
- 將方法移至 SalesData interface 中
  - default 方法
  - static 方法

# 介面 default 方法

- Java 8 介面中可定義 default 方法
  - 使用 default 關鍵字修飾方法。
    - 擁有實作內容
  - default 方法可被實作子類別的物件使用
  - 事後加入不影響子類別
    - default 方法已經有內容，不會強制子類別必須實作該方法
  - 非 default 方法不能有 {}

# default 方法

```
public interface SalesData {  
    public String getName();  
    public double getTotalPrice();  
    public double getTotalCost();  
    public double getProfit();  
    public default void show() {  
        System.out.println("== " + this.getName() + " ==");  
        System.out.println("Price: " + this.getTotalPrice());  
        System.out.println("Cost: " + this.getTotalCost());  
        System.out.println("Profit: " + this.getProfit());  
    }  
}
```

```
public class ProductTest2 {  
    public static void main(String args[]) {  
        SalesData[] itemList = new SalesData[5];  
        //SalesTool tool = new SalesTool();  
        itemList[0] = new Rock(15.0, 10.0, 50.0);  
        itemList[1] = new Rock(11.0, 6.0, 10.0);  
        itemList[2] = new Paint(13.0, 8.0, 25.0);  
        itemList[3] = new Widget(7.0, 5.0, 10);  
        itemList[4] = new Widget(12.0, 12.0, 20);  
        System.out.println("*** Sales Report ***");  
        for (SalesData item : itemList)  
            item.show();  
    }  
}
```

# static 方法

## ■ Java 8 介面中可定義 static 方法

- 以介面名稱呼叫類別方法
- 允許有實作內容
- 傳入要處理的物件

```
public interface SalesData {  
    ....  
  
    public static void show(SalesData item) {  
        System.out.println("== " + item.getName() + " ==");  
        System.out.println("Price: " + item.getTotalPrice());  
        System.out.println("Cost: " + item.getTotalCost());  
        System.out.println("Profit: " + item.getProfit());  
    }  
}
```

```
public class ProductTest3 {  
    public static void main(String args[]) {  
        SalesData[] itemList = new SalesData[5];  
        itemList[0] = new Rock(15.0, 10.0, 50.0);  
        itemList[1] = new Rock(11.0, 6.0, 10.0);  
        itemList[2] = new Paint(13.0, 8.0, 25.0);  
        itemList[3] = new Widget(7.0, 5.0, 10);  
        itemList[4] = new Widget(12.0, 12.0, 20);  
        System.out.println("*** Sales Report ***");  
        for (SalesData item : itemList)  
            SalesData.show(item);  
    }  
}
```

---

# 課程大綱

- 1) 介面進階
- 2) **Design pattern**
  - **DAO 設計模式**
  - **Factory 設計模式**

# 設計模式 Design Pattern

## ■ 設計模式

- ❑ 軟體開發中共通的問題/需求,重複使用的解決方案
- ❑ 經典書籍：Design Patterns: Elements of Reusable Object-Oriented Software (by Erich Gamma et al.)
- ❑ 討論物件導向程式設計的共同溝通語言

# 設計原則

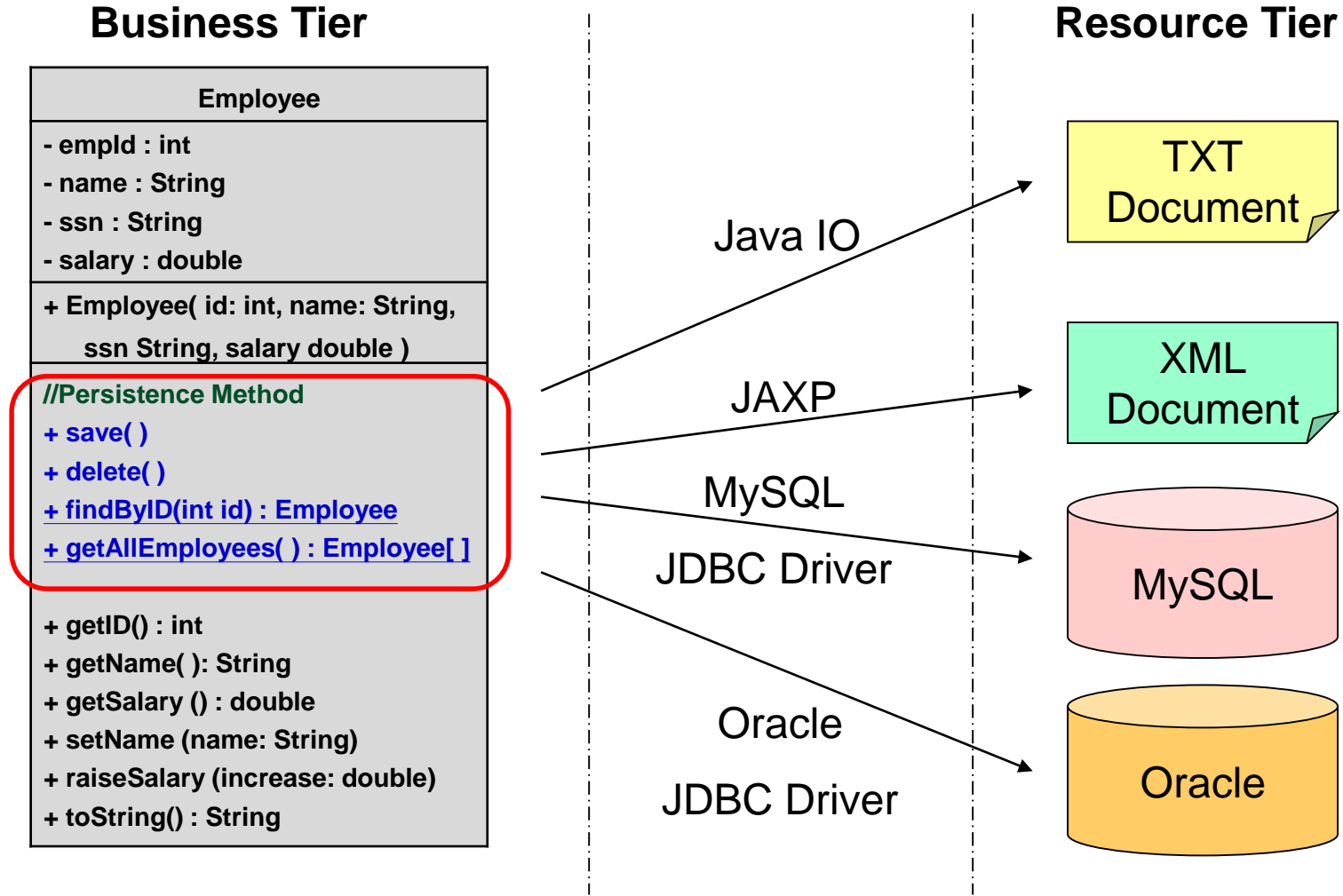
- 物件導向程式設計原則
  - Program to an interface, not an implementation
- 常用interface 相關 Design Pattern
  - Data Access Object (DAO) Design Pattern
  - Factory Design Pattern



# DAO Design Pattern

- DAO用於使用永續性資料的應用程式
- Problem
  - 開發企業應用系統時,有不同方式儲存永續性資料
    - 記憶體 (非永續性)
    - 純文字檔
    - XML文件
    - 關聯式資料庫(RDBMS)
    - Java Persistence API
  - 不同資料儲存型態有不同的存取方式
  - 資料儲存型態改變,程式需大幅修改,不易維護

# Before DAO Pattern



# DAO Design Pattern

## ■ Data Access Object 資料存取物件

- 將資料存取相關操作封裝成DAO一個物件
  - 建立資料連結
  - 執行資料處理
- 置於商業邏輯與資料庫資源之間
- 解構(decouple)業務邏輯與資料庫資源的耦合性

## ■ 實作方法

- 使用interface定義永續性資料的存取方法
- 不同永續性資源(記憶體/檔案/關聯性資料庫)的DAO物件, 提供各自的實作
- 商業邏輯元件由interface的角度來操作DAO物件

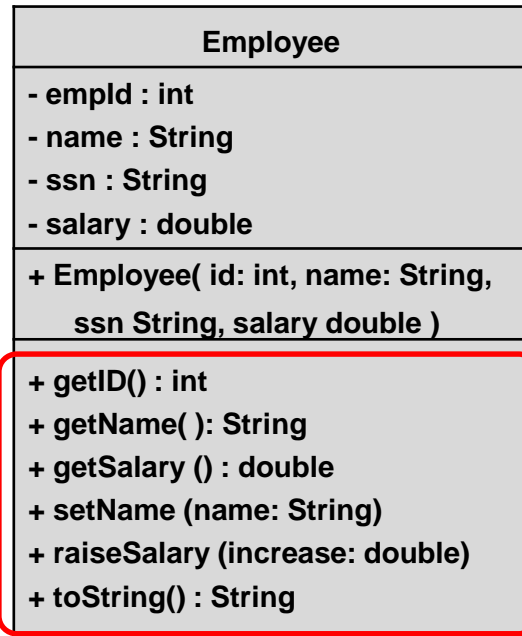
# DAO Design Pattern

## ■ 優點

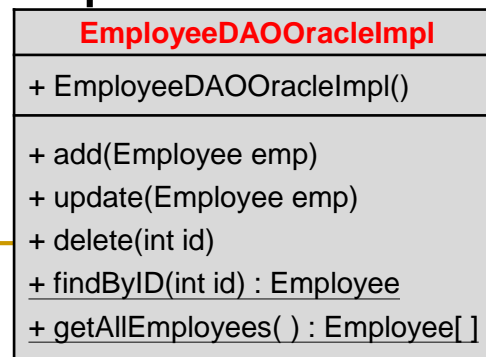
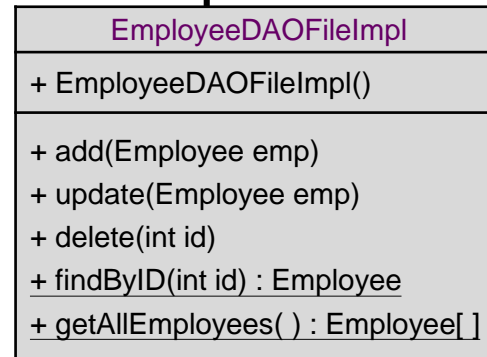
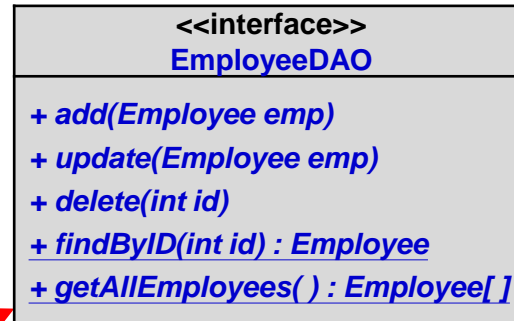
- ❑ 將商業邏輯與資料存取邏輯分離
  - 商業邏輯元件改變,不影響資料存取元件
- ❑ 簡化存取多種資料來源的方式
  - 當資料來源改變,只需修改DAO物件,不影響商業邏輯元件
  - 不同資料來源撰寫獨立的DAO物件,提供一致的資料存取方法
- ❑ 資料存取邏輯可重複使用
- ❑ 集中管理資料存取邏輯

# DAO Design Pattern

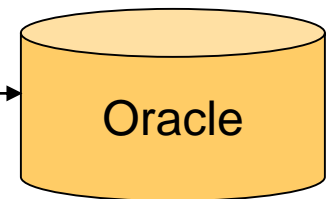
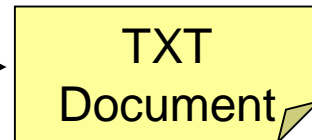
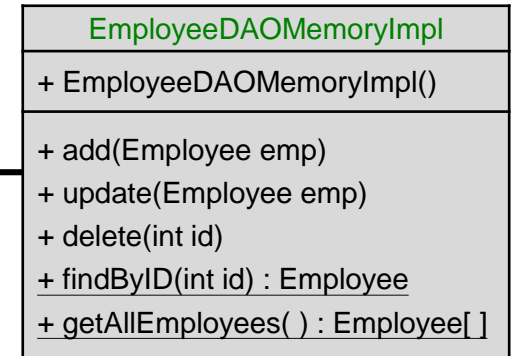
## Business Tier



## Integration Tier



## Resource Tier



EmployeeDAO dao = new  
EmployeeDAOMemoryImpl();

EmployeeDAO dao = new  
EmployeeDAOFileImpl();

EmployeeDAO dao = new  
EmployeeDAOOracleImpl();

Java IO

Oracle  
JDBC Driver

# 課程大綱

1) 介面進階

2) **Design pattern**

- DAO 設計模式
- **Factory** 設計模式

# Factory Design Pattern

```
EmployeeDAO dao = new EmployeeFileDAOImpl();
```

## ■ Problem

- ❑ 使用Interface操作DAO物件,資料來源改變時,後續程式不受影響
- ❑ 使用建構子建構DAO物件,資料來源改變時,建構式程式碼需修改
- ❑ 通常會有多處DAO建構式散佈在應用程式中,造成維護的困難

# Factory Design Pattern

## ■ Factory 工廠模式

- 以工廠方法取代建構式呼叫,來建立**DAO**實作物件
- 解構(decouple)商業邏輯與**DAO**特定實作的耦合性

## ■ 實作方法

- 撰寫一個用來建構**DAO**實作物件的工廠類別
- 工廠類別中提供建構**DAO**實作物件的工廠方法
- 商業邏輯元件先建構工廠物件,再呼叫其工廠方法,來建立**DAO**實作物件



# Factory Design Pattern

